TEC-0112

Prototyping the DARPA Image Understanding Environment and Tools to Facilitate Its Use

Daryl T. Lawton, et al.

Georgia Institute of Technology
GVU Center, College of Computing
Atlanta, GA 30332-0280

August 1998

Prepared for:
Defense Advanced Research Projects Agency
3701 North Fairfax Drive
Arlington, VA 22203-1714

Monitored by:
U.S. Army Corps of Engineers
Topographic Engineering Center
7701 Telegraph Road
Alexandria, Virginia 22315-3864

US Army Corps of Engineers
Topographic Engineering Center

T

E

C

Destroy this report when no longer needed.
Do not return it to the originator.

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

The citation in this report of trade names of commercially available products does not constitute official endorsement or approval of the use of such products.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	August 1998	Annual Technical September 1992 - August 1993

4. TITLE AND SUBTITLE

Prototyping the DARPA Image Understanding Environment and Tools to Facilitate Its Use

5. FUNDING NUMBERS

DACA76-92-C-0016

6. AUTHOR(S)

Daryl T. Lawton, Warren Gardner, David Dai, Ted Rathkopf, Shumei Lin, Heather Prichett, Jan Smith, Yong Wei Yang, Mary Ann Frogge, Jun Hoy Kim

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Georgia Institute of Technology
GVU Center, College of Computing
Atlanta, GA 30332-0280

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Defense Advanced Research Projects Agency
3701 North Fairfax Drive, Arlington, VA 22203-1714

U.S. Army Topographic Engineering Center
7701 Telegraph Road, Alexandria, VA 22315-3864

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

TEC-0112

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This report is a summary of the first year's research on Prototyping the DARPA Image Understanding Environment (IUE) and Tools to Facilitate Its Use. The major objectives of this project are to support the design and development of the IUE, to prototype the IUE user interface and data exploration tools, and to develop tools for documentation, tutorials, and publication. During this first year of the project, the design of the IUE user interface has been completed and prototypes have been developed for testing and evaluating the interface. Authoring tools for on-line documentation also have been developed.

14. SUBJECT TERMS

Technology transfer, annotation, interface design, prototyping

15. NUMBER OF PAGES
123

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UNLIMITED

Table of Contents

List of Figures

List of Tables

PREFACE

This research was sponsored by the Defense Advanced Research Projects Agency (DARPA) and monitored by the U.S. Army Topographic Engineering Center (TEC) under contract DACA76-92-C-0016. The DARPA Program Manager was Mr. Oscar Firschien, and the TEC Contracting Officer's Representative was Ms. Lauretta Williams.

Chapter 1

1.1 Project Overview

This section provides an overview of the objectives and major results from the first year of the project and provides an introduction to the entire report. The DARPA Image Understanding Environment (IUE) is a common software environment for image understanding research which will support the transfer of technology and research from the DARPA Image Understanding community. The major objectives of this project are to support the design and development of the IUE; to prototype the IUE user interface and data exploration tools; and to develop tools for documentation, tutorials, and publication which will facilitate the impact and the widespread adoption of the DARPA IUE.

The primary concerns of the work on this project are:

- The overall design of the IUE through activities of the IUE Technical Committee ("The Gang of Ten"),
- Prototyping the IUE user interface and data exploration tools,
- Developing tools for documentation, tutorials, and publication which will facilitate the impact and the widespread adoption of the DARPA IUE,
- Basic Vision research by students supported by the contract.

The first two sections of this report present the design and initial prototyping of the user interface of the DARPA IUE and the tools being developed for documentation, tutorials, and publication that will facilitate the use and adoption of the IUE. The following chapters contain presentations of motion processing and visual navigation algorithms. These include generalizing earlier work for processing translational image sequences to less restricted motions; extensions to factorization methods to allow for linear features which are less dependent on precise feature-point matching; the incorporation of models in processing dynamic images; and algorithms for range-free qualitative navigation which enable mobile robots with limited recognition capabilities to form effective spatial maps for navigation and exploration.

1.1.1 First Year Accomplishments

- Completed the design of the IUE user interface.
- Prototyped essential parts of the IUE interface for testing and evaluation.
- Explored potential target interfaces and graphics packages for implementing the IUE.
- Developed authoring tools for on-line documentation of the IUE and the presentation of tutorials.
- Researched areas such as dynamic image processing, robot navigation, and interactive model based vision systems.

1.1.2 Objectives for Second Year

Next year, 1994, is a critical time for the overall success of the IUE project. This project is expected to have initial versions of the object hierarchy and an implementation of the data exchange format to begin working with. The objectives are:

- Complete all interface prototyping activities and make the implementation publicly available.
- Complete development of prototype hypermedia annotation tools for on-line IUE documentation and tutorials.
- Complete evaluation of publicly available hypertext systems for on-line IUE documentation and tutorials.
- Develop prototype tutorials using hypermedia annotation tools or publicly available hypertext systems.
- CD-ROM version of DARPA IU Workshop Proceedings.
- Organize IUE Tutorial Materials for different books and on-line resources.
- Continue research in motion processing and interactive model-based vision systems.

1.1.3 Plans for Final Year

The primary objective of the final year of the contract will be to produce an on-line tutorial introduction to the IUE and machine vision, and to integrate these tools with existing publishing mechanisms for the IU Workshop. Research in terrestrial motion processing and interactive model based vision systems will also continue.

1.2 IUE Interface Design and Prototyping

The intent of the IUE user interface is to provide exploration tools and to interact with IUE objects. Therefore, it is being developed in three levels: the graphics level to tell the screen what to do, the interface support level to create and prototype interfaces and related tools built on top of the graphics level software, and the image understanding environment user interface (IUEUI) level which is a specialized interface for image understanding. Chapter 2 will discuss these interfaces in more detail.

1.2.1 Object Displays

Object Displays are used for viewing and interacting with objects by mapping them onto a two-dimensional display window. This involves nearly all IUE objects: images, curves, regions, object models, surfaces, vector fields, etc. Object displays support several types of operations for controlling the mapping of an object onto a window, such as the viewing transformation, mapping values through pixel-mapping functions and color look-up tables, the specification of overlay planes, transparency effects, interacting with displayed objects through selection operations, and interactive function application.

There are different types of object displays:

- The **image** display is for viewing images and image-registered features.
- The **local graphics** display is for displaying objects by mapping their values onto parameterized graphic objects such as lines and cubes. Examples are displaying vector fields and edges.
- The **surface** display is for displaying objects that get mapped onto mesh or rendered surfaces.
- The **plot** display is for displaying functional relations between objects. Examples are one-dimensional, two-dimensional, and three-dimensional graphs, histograms, scattergrams, and views of functions and tables.

These different types are distinguished by specific methods but all inherit a large number of similar methods from the general display class. For example, overlay operations are similar for a surface display and for an image display, although they can look quite different. (In one case it appears as a drawing in solid colors in image-registered coordinates on top of a displayed image, and in the other it renders the colors onto a displayed surface.) Plot displays have many similarities with object displays in terms of such things as overlays and interaction methods.

Chapter 2 will also discuss browsers, which interact with text-based or symbolic descriptions of objects. Field Browsers consist of a regular array of fields. Fields can be filled with text, icons, colors, colored text, or text in particular fonts. Fields can have actions associated with them when they are selected or a user changes the values in them. Chapter 2 also includes information on command buffers and command languages; contextual descriptions of the state of the interface and the status of displayed IUE objects; and prototyping the many different parts of the user interface to complete the functional specification and to answer basic implementation questions.

1.3 IUE Documentation and Tutorials

The IUE will be supported by on-line documentation and tutorials which will facilitate the impact and the widespread adoption of the DARPA IUE. The tools for implementing these will also be available for enhanced communication and publication by scientists and developers who use the IUE. While there is significant activity in developing documentation and hypermedia toolkits, they remain largely machine dependent with no clear standardization.

Work on this project during the first year went through three distinct phases. First Lucid Emacs 19, was adapted to have some hypertext features. Then annotation capabilities were implemented using a Tk/Tcl. Finally, MOSAIC and HTML became useful documentation and tutorial delivery mechanisms because of their widespread and growing use throughout the educational and scientific community.

The work described in Chapter 3 concerns the prototype hypermedia annotation system currently being developed is called "Knowledge Weasel" (KW). It is a presentation and authoring system designed to support annotation using several different types of media. A simple analogy for KW is reading a book or attending a lecture and being able to make diverse types of comments and annotations on the material. In reality, such unrestricted annotations and comments made with respect to real books and lectures could create a significant mess (especially if made by several different people), so in developing KW this simple metaphor was extended in several ways. The first provides a general format for annotations that can include several different types of media. An annotation is a common record structure wrapped around instances of different types of media such

as text files, sound, drawings, postscript files, GNU-plots, code running in the GDB debugger, and others. Annotations are implemented in the same way as property lists in Lisp, with attributes and values, and are displayed as buttons with an associated region of support. When an annotation is selected, it performs an operation specific to the type of annotation selected. Annotations are created using existing media editing tools for operations such as recording a sound, drawing packages, calls to other branched processes, and grabbing a portion of the screen. The second extension, has been to develop, different types of navigation, organization and presentation tools to keep users from being overwhelmed with a great deal of possibly irrelevant information. Users can prune the set of annotations that they want to deal with and also how these annotations are displayed. Annotations are structured to make possible intelligent processing, perhaps eventually including rule-based processing, for automatic presentation and "ferreting" of information (hence the name). An important commercial use of computer vision technology in the near future will be adapting model-based vision technology so users can interactively annotate images with models.

Implementation of KW on top of Lucid Emacs 19, which is in turn based on the X window system, began. Lucid's implementation of Emacs Lisp provided primitives for handling display attributes such as windows, fonts, and colors, and has a built-in Lisp interpreter for Emacs Lisp. This Lisp variant provides a wide variety of primitives that are useful for manipulating text, processes, and/or files. Current implementation is in Tcl/Tk.

1.3.1 CD-ROM version of DARPA IU Workshop Proceedings

A significant instance of technology transfer is the DARPA IU Proceedings and Workshop. For the next meeting, plans are to enhance this transfer by having the workshop proceedings available on CD-ROM and integrated with the Data Exchange Format, a documentation and browsing tool such as Knowledge Weasel, and, possibly, the IUE itself. This will enable an extraordinary type of paper which includes data, code, additional references, animations, extensive annotations and cross-references.

The second major part of this project is to develop authoring tools for producing documentation, demonstrations, and tutorials. These will be used for on-line documentation of the IUE and to support publication of research. The authoring tools being developed are based upon existing hypermedia and interface construction kits. The authoring tools and data exploration tools will be used to develop an interactive, on-line tutorial for learning how to use parts of the IUE.

Currently, the yearly proceedings from the DARPA IU Workshop are a major source of technical output from the IU community. The IUE and the modules developed in this project will extend this significantly. People will have access to a much wider range of information than is currently contained in published proceedings. This will include such things as code, data, slides, viewgraphs, and tutorials developed by the authors themselves available on-line and through CD-ROM.

1.4 Translational Decomposition of Flow Fields

Chapter 4 presents a set of algorithms for processing optic flow fields by approximating them as local translations of the corresponding portions of the environment. This is theoretically interesting since it dramatically simplifies the equations for inferring motion parameters from optic flow and also supplies a low level representation of image motion that might be useful for inferring motion properties from non-rigid motions. Its practical use involves its robust nature for motion constrained to an unknown plane which characterizes much of terrestrial robotics. It can also use a small number of points for inferring motion parameters from an optic flow field.

Once the directions of motion have been established, they can then be used as constraints to determine the actual parameters of motion and to recover the structure and layout of environmental surfaces. Motion direction is broken into four different cases: (1) motion constrained to a known plane (the normal to the plane is known); (2) motion constrained to an unknown plane (the normal is not known); (3) motion constrained to surfaces which are locally planar; and (4) arbitrary motion with no assumptions.

1.5 Interactive Model Based Vehicle Tracking

While most work in motion processing has involved very minimal assumptions about objects such as rigidity, a very important area for future work is motion processing which incorporates object models. Investigation has been started in the restricted domain of tracking vehicles from a stationary camera in outdoor road scenes. The key idea is that motion is a critical source of information for instantiating object models and that motion processing is in turn simplified by the constraints supplied by object models.

Processing begins with a human forming a rough interpretation of a scene by interactively manipulating models of objects such as terrain surface patches, roads, gravity, and vehicles. This initial, human-directed interpretation consists of incompletely specified two dimensional drawings of expected image features and associated three-dimensional object models which are also initially incompletely specified. Once an interpretation is in place, tracking algorithms then autonomously refine and extend the interpretation. For example, a human will indicate that a particular area is a road by a two-dimensional drawing. The system will then track movement along the road and fit a constraint-based description of a vehicle to this movement. As vehicles are tracked, the three-dimensional shape of the road can be recovered. The system can determine that a vehicle has just gone off the road (or that it is behaving inconsistently with respect to the model of a vehicle) and report back to a human about unusual occurrences or behavior for which it cannot account.

Object models are related by constraints specifying necessary geometrical properties and relationships between objects. The use of constraints allows for flexible object instantiation. A user can indicate a vehicle and direct perceptual processing routines to determine the corresponding local surface orientation and roads, or he can instantiate a road segment to direct the extraction and tracking of vehicles.

The work with the local translational approximation described above has been found to be useful for tracking vehicles and determining three-dimensional information. Moving vehicles can often be treated as rigid objects which are translating over short periods of time. For example, as a vehicle goes around a curve, because of turning radii constraints, the axis of rotation is often far away from the vehicle itself and the vehicle motion can be treated as a sequence of small translations corresponding to tangents of the curve of motion. The local translation-based tracker determines the direction of motion of a set of extracted image points over time, and fits their motion to an estimate of the current direction of motion of the corresponding vehicle in three dimensions. The effect of this tracker can be visualized as a unit sphere with an axis corresponding to the current direction of motion. As the vehicle and the corresponding set of points move, the position of the axis changes with respect to the sphere. This processing is expected to work well with temporal filters since there are constraints on how quickly a vehicle can change its direction of motion. Vehicle rotation is indicated by areas of the image which show differences over time, but for which no clear axis of translation can be determined. Conversely, if there is an instantiated three-dimensional road model and a rough estimate of the position of the vehicle along the road has been established, the tangent information associated with the road model can be used to initialize the search for the axis of translation. If there is an instantiated vehicle model, it restricts the features that the local translational tracker uses.

This work will be useful for applications such as telerobotic monitoring systems where low bandwidth communication is critical. The human would produce a rough scene interpretation from sensory information from a telerobot. The resulting interpretation is a model of the world that the telerobot would refine, use to control its behavior, or report back to a human. In this way, the human directs the telerobot by initializing and constraining its processing. Communication between the robot and the human takes place in the context of a shared model of the world which makes possible infrequent, semantically meaningful, and very low bandwidth communication.

1.6 Shape and Motion from Linear Features

The extraction of environmental structure and motion from a sequence of two-dimensional images is a common problem in computer vision. Research is attempting to overcome the disadvantages associated with a camera-centered representation using a world-centered coordinate system to compute shape and motion without the intermediate calculation of depth. This work is discussed in chapter 6.

1.7 Range-Free Qualitative Navigation

Qualitative Navigation [13, 20] concerns spatial learning and path planning in the absence of a single global coordinate system for describing locations and positions of landmarks. It is based on a multi-level representation of space which, at its most abstract level, is based on topological properties which allow a robot to describe a location using the directions of visually salient patterns (with no associated range measurements) and then navigating using the occlusions that occur among them as a basic cue to control movement through the environment. This work [17,16] in qualitative navigation was developed while trying to produce basic navigation and recognition capabilities in an autonomous land vehicle. Chapter 7 describes qualitative navigation algorithms which work completely at the topological level, dealing with landmarks for which there are no range estimates.

Chapter 2

Prototyping the IUE User Interface

2.1 Introduction

The user interface of the IUE is intended to provide flexible, simple, and powerful tools for exploring data, algorithms, and systems. The general principles of object-oriented design used in developing the IUE object hierarchy and programming constructs have also been applied to the interface: abstraction over common operations to provide a small number of interface objects that can be freely combined by a user. The interface has been designed to have a consistent interaction with IUE objects and their semantics, especially the abstraction in the IUE object hierarchy. Thus, the display and browsing operations are sensitive to the class similarities for objects such as images, image-registered features, and spatial objects. Using and becoming comfortable with the interface should not involve understanding a large number of unrelated things.

An equally important part of the user interface is what it does not develop. The IUE user interface must leverage extensively off of existing (and emerging) interface and graphics packages and standards. The interface must be supported by ongoing and future developments in software environments and graphical user interfaces. This is critical for the long term use of the IUE because of continuous advances in these areas which would be advantageous in terms of capabilities and cost.

To realize this, the interface is being developed in terms of three levels (figure 2.1). The **Graphics Level** is the underlying "machine independent" package for display and graphic operations which tell the screen what to do. Examples would be X, GL, OpenGL, and Phigs. The **Interface Support Level** involves packages to create and provide rapid prototyping user interfaces and related tools, which are built on top of graphics level software. This also includes the tools found in the selected software development environment, such as editors and debuggers. The **Image Understanding Environment User Interface (IUEUI) Level** consists of the interface objects specialized for image understanding. This includes such things as object displays, plotting displays, several types of browsers, and structures for describing the interface context. The IUEUI consists of a small set of objects that can be freely combined for very powerful results. The specifications of these objects are relatively independent of the other two levels, although much of the current prototyping and design activities are directed towards understanding how to best realize the functionality of the IUEUI objects with respect to these two levels, especially for accessibility and limiting the eventual cost of the IUE for users.

The basic functional components of the IUE interface, as depicted in figure 2.2, are:

- **Displays**: These deal with mapping spatial objects and images (or sets of spatial objects and images) onto two-dimensional display windows. There are several types of displays for displaying images and image-registered features, for plotting functional relations between attributes and components of spatial objects, and for displaying surfaces.

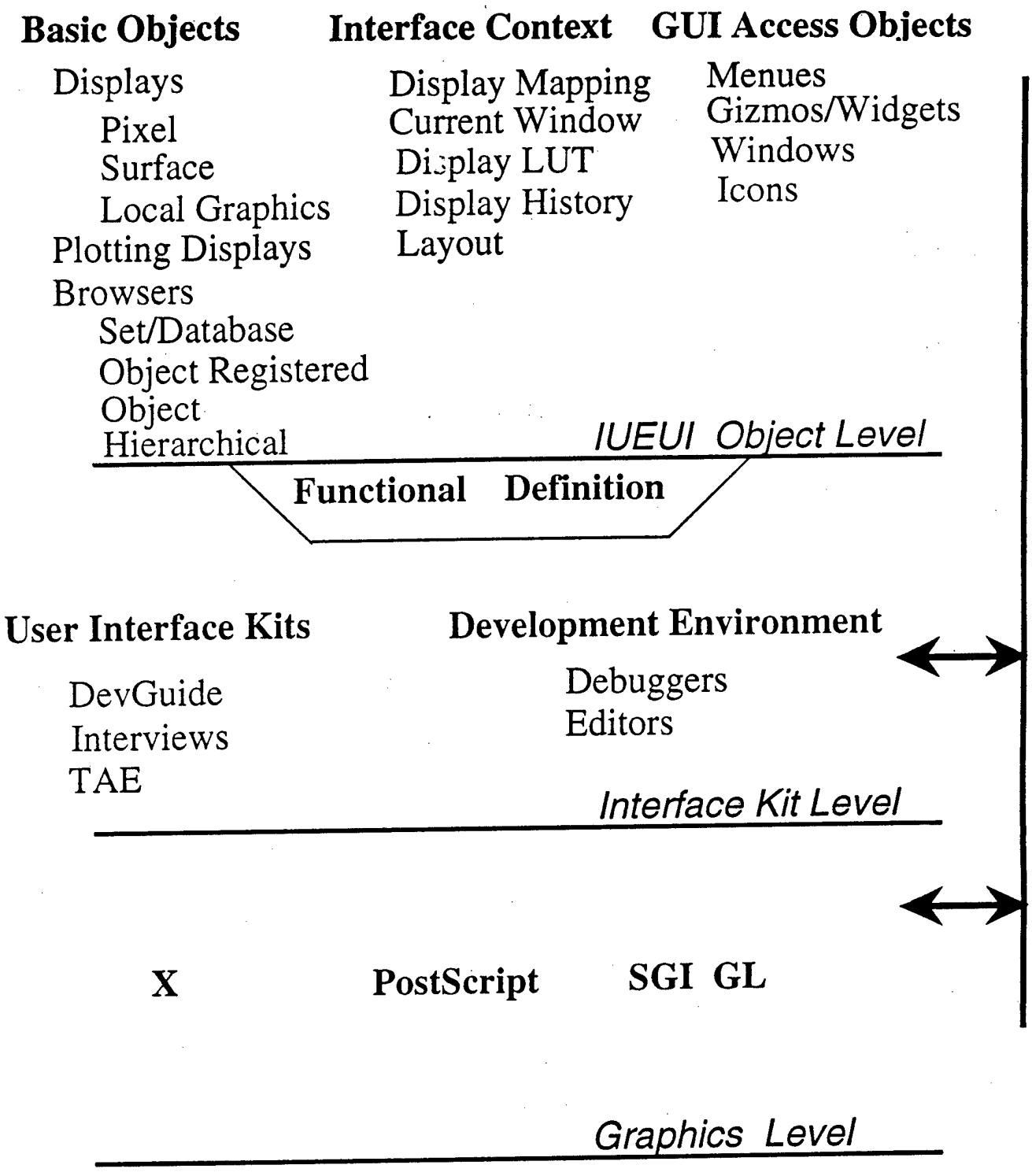

Figure 2.1. Interface Levels.

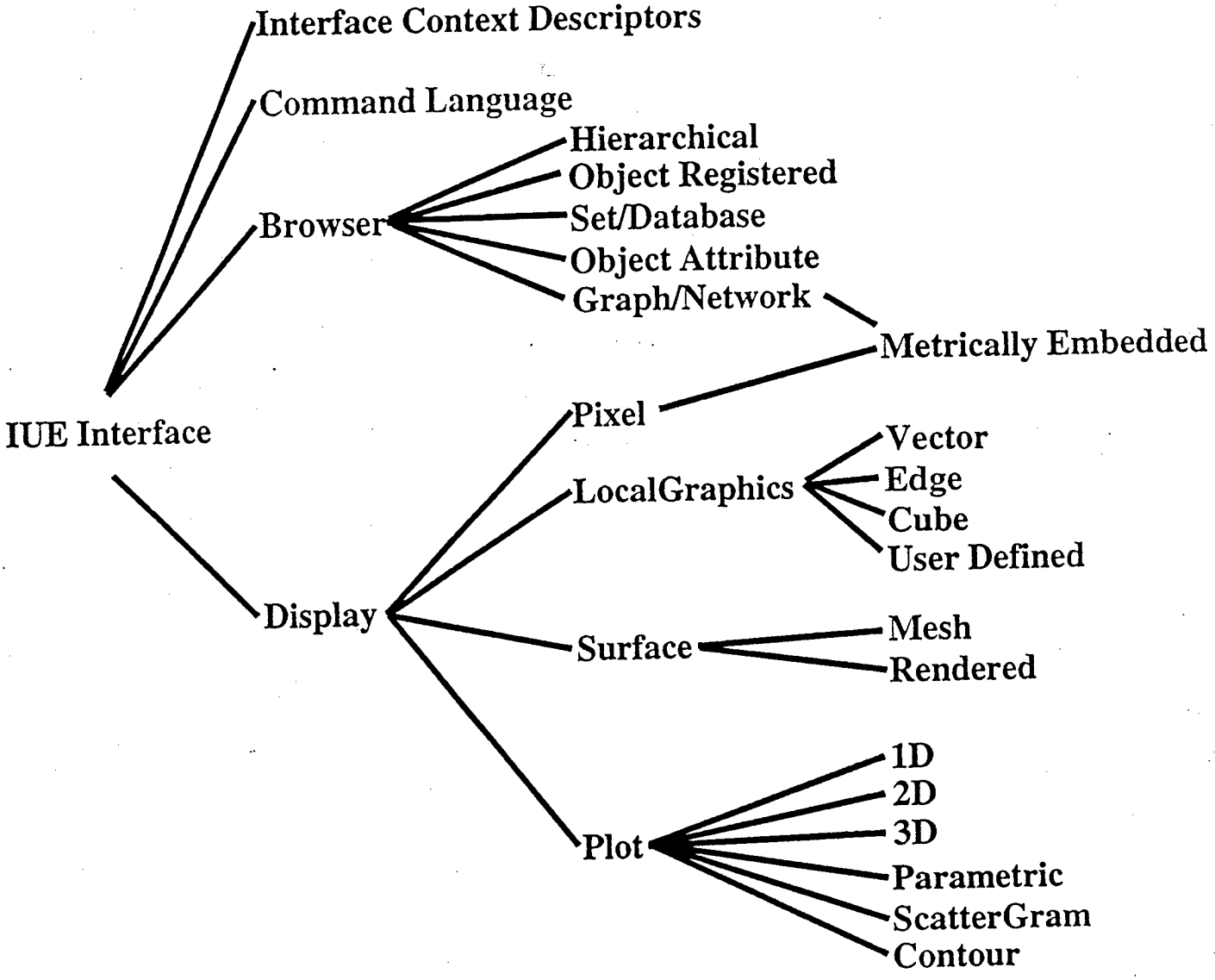

Figure 2.2. Interface Functional Components.

- **Browsers**: These deal with presenting textual and symbolic information about objects. There are different types of browsers for performing operations, such as inspecting the values in a spatial object, for performing interactive queries with respect to databases and sets of objects, and for inspecting relational graphs and networks.

- **Interface Context Descriptors**: These are for describing the state of the interface and interface objects. Examples are such things as the current color-look-up table for a given display, the current display window or browser, and links between interface objects which describe related views. This information supports intelligent default behaviors.

- **Command Language and Command Buffer**: Users can control their interaction with objects by using an interactive command language. The commands can be used in code and to create scripts. This also provides a complete description of the functionality of the user interface.

- **Simplified, programmable access to Graphical User Interface (GUI) objects**: This is intended to provide programmer access to several of the objects commonly found in GUI Construction Kits, such as knobs, sliders, text buffers, and menus. These can then be used in applications and to extend the interface.

2.1.1 Prototyping

Prototyping many different parts of the user interface to complete the functional specification and to answer basic implementation questions about choices regarding GUIs and user interface toolkits began. Prototyping of the interface went through three distinct phases. First, mock-ups of the different interface objects were developed using the Interface Builder on the NeXT machine. This allowed for rapid prototyping of the objects for look-and-feel. For reasons of rapid development, the next step began implementation in C on Silicon Graphics (SGI) machines using the GL graphics library, Motif, and the FORMS user interface toolkit. Using these interfaces, the general display object and the different browsers were put up very quickly. As part of this, extensions to GNUPlot are currently being explored to make sure it is compatible with methods associated with the general display class and so it can provide an inexpensive plotting package. OPENGL is also being evaluated as a possible machine-independent graphics package to provide the powerful functionality of the SGI graphics library and FORMS. In the third stage of prototyping, implementation on the IUE objects in Tk/Tcl in C++ has begun. This is a very rich machine-independent, toolkit for developing interactive user interfaces using an object-oriented language.

An example of an early mock-up done on the NeXT machine is shown in figure 2.3.

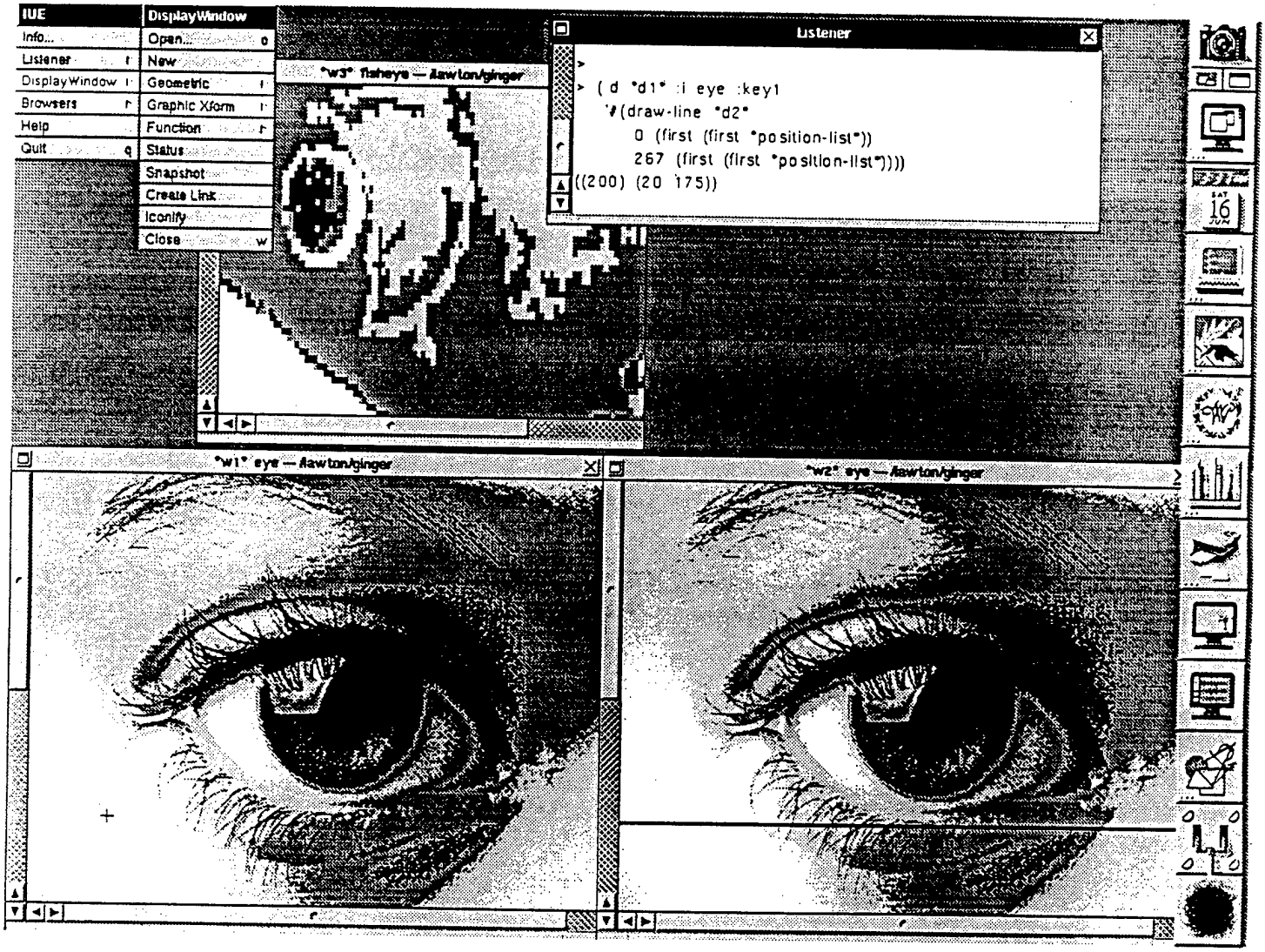

Figure 2.3. NeXT Mock-Up of IUE User Interface.

For comparison, the initial form of the graph browser using the NeXT Interface Builder is shown in figure 2.4.

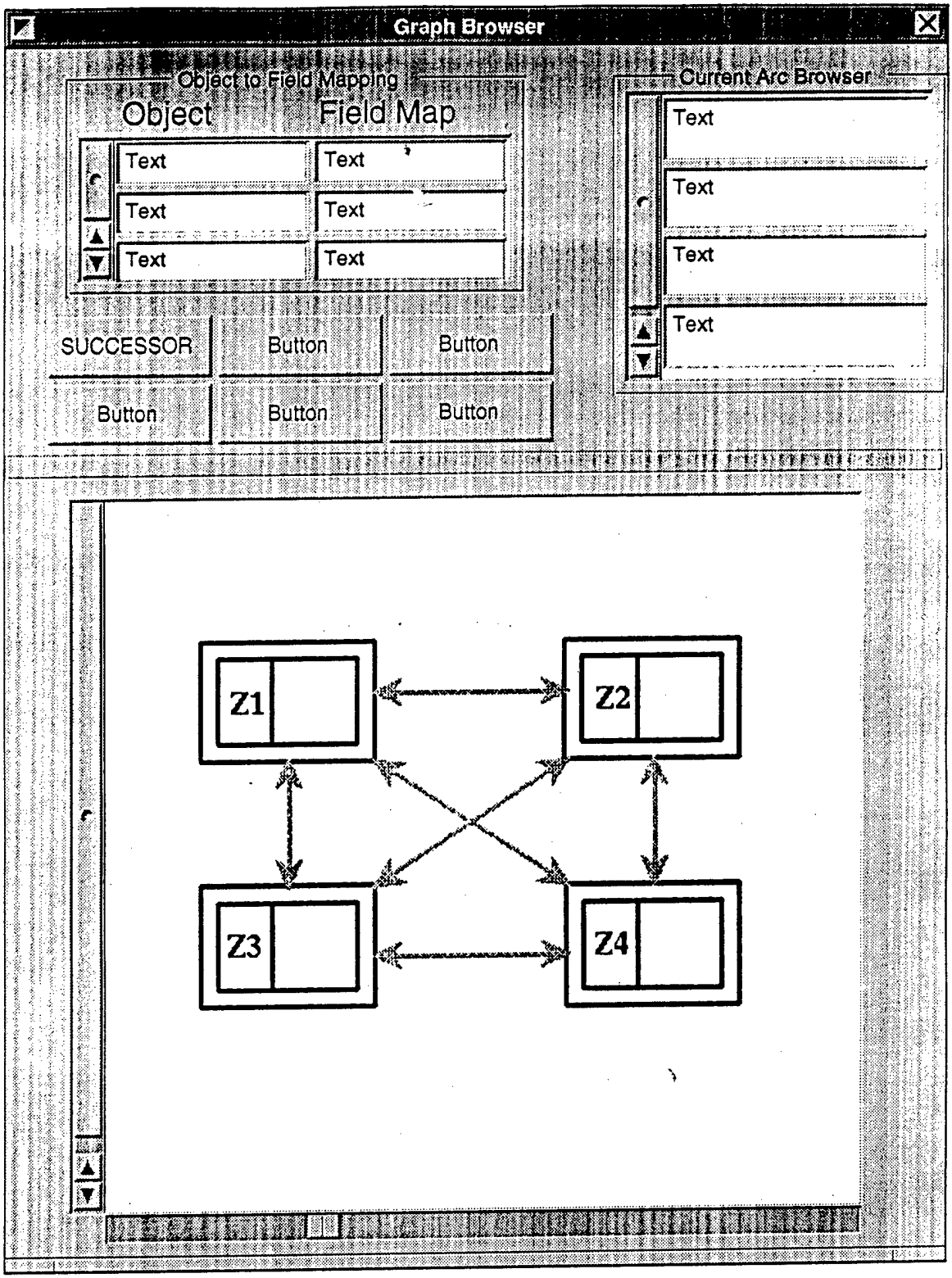

Figure 2.4. NeXT Mock-Up of Graph Browser.

The implementation of the graph browser, using the FORMS interface kit on SGI machines is shown in figure 2.5.

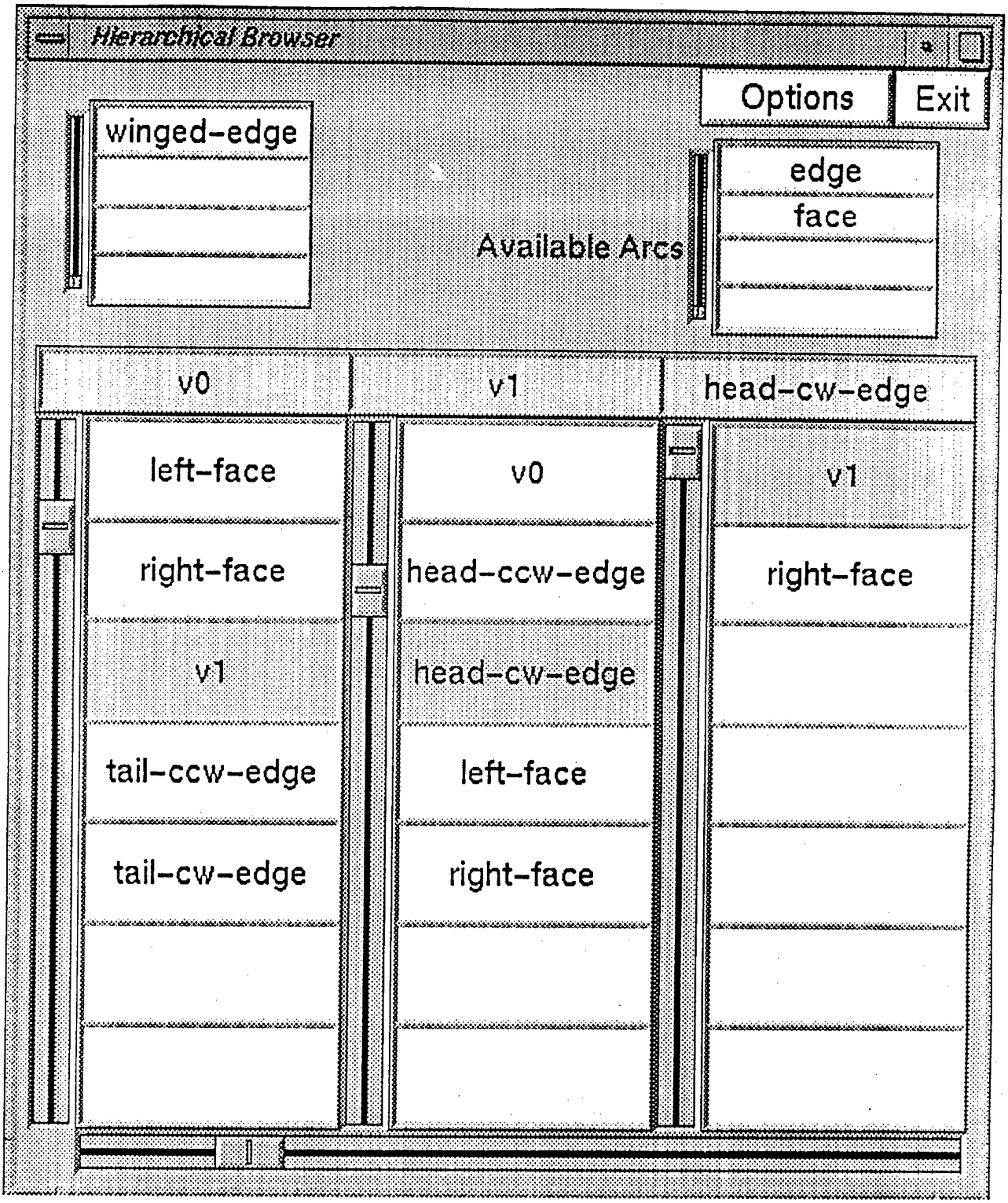

Figure 2.5. SGI FORMS Implementation of Graph Browser.

The current implementation, using Tcl/Tk is shown in figure 2.6.

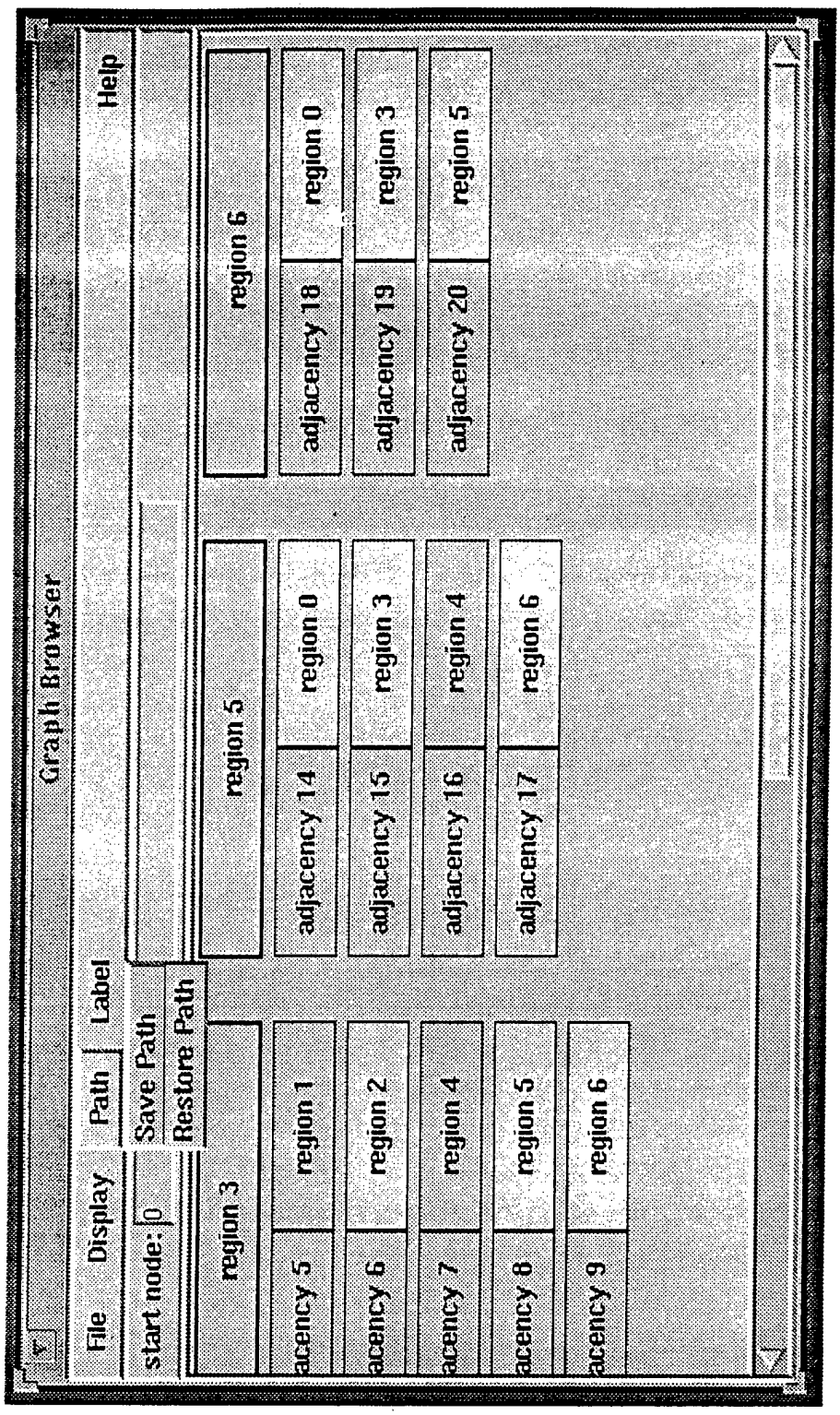

Figure 2.6. Tcl / Tk Implementation of Graph Browser.

With the exception of GUI objects, we can review each of these in more detail. Our focus here is on the functional components of the user interface and its attributes. The examples are from prototyping on the SGIs using the FORMS Interface Construction Kit.

2.2 Object Displays

Object Displays are used for viewing and interacting with objects by mapping them onto a two-dimensional display window. This involves nearly all IUE objects: images, curves, regions, object models, surfaces, vector fields, etc. Object displays involve a wide range of actions, such as displaying an image and image-registered features; displaying networks of objects, such as stereo images, multi-resolution pyramids, image sequences (and this can involve having several linked windows for the different images, cycling through displays of the different components, or mapping the different components onto different planes of the display buffer and combining the images through transparency or color addition); displaying models and predicted segmentations as overlays; and interactively inspecting and manipulating displayed objects and applying operations to them.

There is a strong relationship between spatial objects and displays. Most IUE objects are expressed as relations between sets. In displaying an object, values from one set are used to specify a position in a display window and the corresponding values from another set are used to specify an attribute such as intensity, color, overlay, transparency effect, etc., that is displayed at the position. A basic example is an image where one set is described by the indices of the coordinate system of the image, and the other set is described by the color or intensity values associated with the particular image coordinate. A discrete curve is a mapping from integer indices onto two-dimensional positions with respect to an image coordinate system. Displaying a curve as an overlay on top of an image involves mapping two-dimensional positions along the curve onto window positions using the same transformation that was used for the display of the image. The color/intensity of the display at these points can be based upon registered values associated with the curve (such as an approximated curvature). For example, a user might want to display an intensity image in 8-bits of grey-level intensity, and then display overlay-extracted curves on top of this while displaying curvature values along the curve mapped on to 8-bits of red intensity.

Operations that are involved with specifying the mapping onto a screen position are referred to as **position** methods. These methods involve panning, zooming, perspective transformations, and warping operations. The machinery for this comes directly from the coordinate system transformation methods. The operations that involve mapping onto a particular window value are called the **value** methods, which involve such things as setting up the Color Look-Up Table (CLUT), the point-mapping functions (functions applied to the value at an object position prior to display), the transparency effects, etc.

The basic processing flow for displays is shown in figure 2.7. Displays that take place with respect to a context involve such things as the current object, the current description of the transformation from an object to the display window, the current CLUT, links between IUE objects, and several other attributes. Several display operations involve setting up these contextual attributes. Displaying an object, such as an image or image-registered features, involves iterating over the object and applying the specified position methods to determining the position in a window at which to generate a display and applying the specified value methods. In interactive processing, a selection operation is performed with respect to the display context to return a value at a selected location. Graphics are also done with respect to the display context (the processing flow in figure 2.7 is idealized in several respects). Many display operations don't involve iterating over an object, but manipulate the CLUT and display the buffer directly. Rendered objects, or displays that involve warping or interpolation, are more naturally expressed as iterating over the

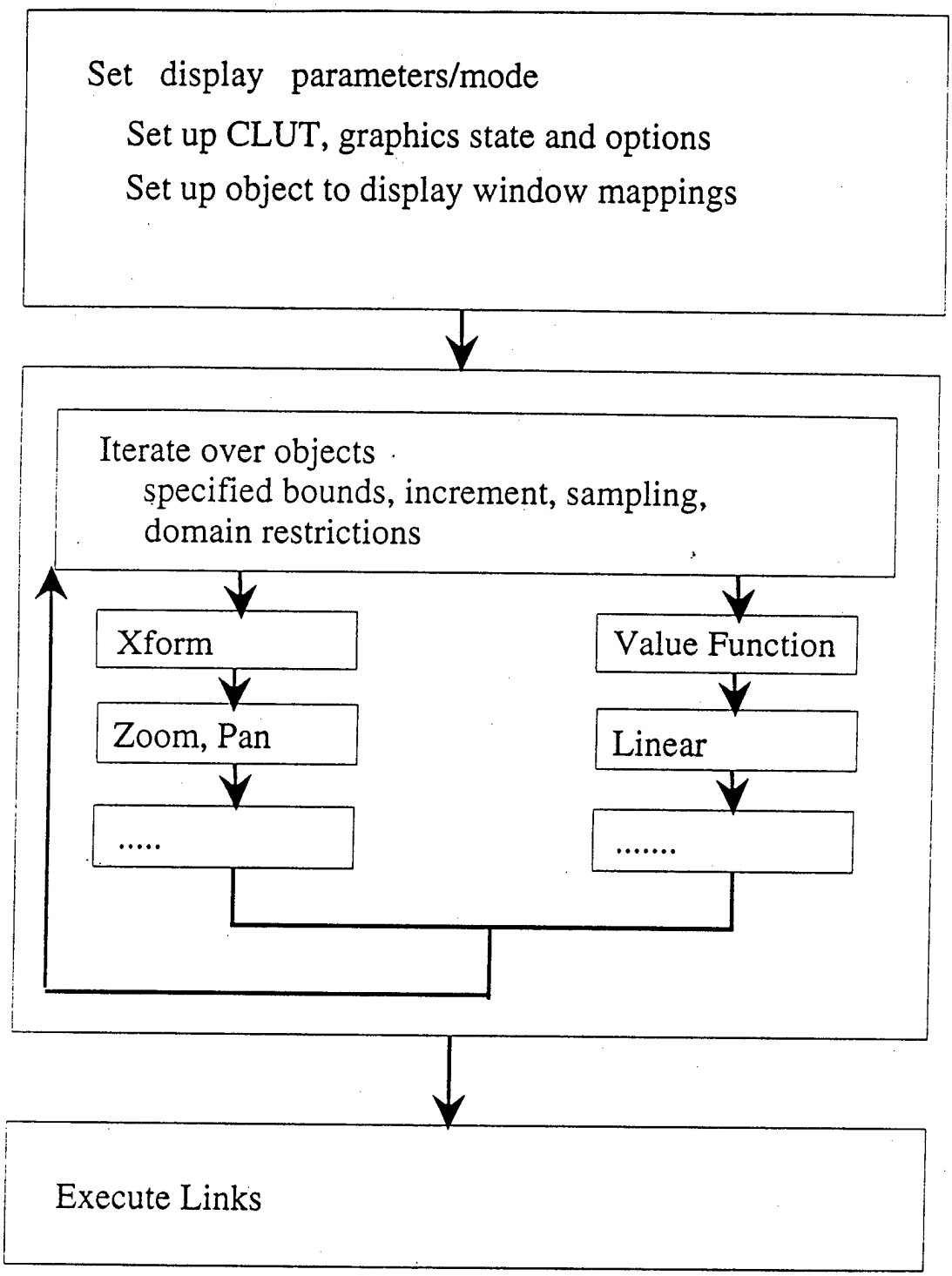

Figure 2.7. Basic Processing Flow for Displays.

display window itself. Displays can also involve a discrete sampling of other objects using square pixel neighborhoods.

The object display methods are organized into the following classes:

Value Methods: These are methods that control how values in the specified object(s) are mapped onto screen attributes, such as color and intensity; how to configure planes in the screen buffer for the display of color images; how many panes to use for overlays; and how to apply particular functions and conditions to object values prior to display. This includes operations such as overlays, mapping onto different color bands, transparencies, and others.

Position Methods: These are methods that control how positions in specified object(s) are mapped onto a display window. This includes operations such as panning, zooming, perspective views, and some types of warping.

Display Window Attributes: These involve controlling attributes of the display window, such as position, size, attributes of the title bar, event handling for the mouse, and re-sizing operations.

Link Methods: These methods are for linking different displays (and browsers and GUI widgets). Examples are window-to-window zooming and displaying stereo and pyramidal images in multiple images. This involves creating links and specifying the operations and transformations associated with links between interface objects.

Interaction Methods: These methods involve interaction and manipulation of displayed object(s) in a display. Operations include selecting objects, recovering object positions and values from a mouse click, and applying functions to selected objects.

Graphics Methods: These are methods to display registered graphics for drawing lines, text, three-dimensional objects, etc.

History Methods: Many objects can be mapped to the same display window. History methods coordinate displays over time, such as cycling through an image sequence or playing an animation of displays.

Archiving Methods: Archiving methods involve printing an object display, writing an object display to file, or writing an object display to video tape.

2.2.1 Value Methods and the CLUT

Value methods are used to specify how values in an object are mapped onto display window attributes such as color and intensity. There are several types of methods for this. **CLUT-segment methods** involve setting up a color look-up table (CLUT) which involves creating named segments and associating some number of bits with each segment, such as specifying 8 bits for red, 8 bits for green, and 8 bits for blue. **CLUT-value methods** involve associating color and intensity values with CLUT indices. These operations can be applied to CLUT segments. For example, the red component of the CLUT can be mapped onto red values in several different ways: by linear interpolation between specified shades of red, or by a spline through specified color values. **Overlay methods** involve setting up overlay planes. Overlay planes can be displayed and cleared separately of underlying intensity displays. **Object-mapping methods** deal with taking object values and mapping them onto color table indices. For example, the CLUT-segment for red intensity could be set for a linearly interpolated 255 shades of red, but the actual object

values in an object could range from values such as -1000 to 2000. The **Linear** object-mapping method specifies to linearly interpolate from this range of object values to the available shades of red. There can be a linear mapping from the object onto color table indices, but the color table may be set up for a non-linear mapping onto actual intensities displayed on the screen. **Value-function methods** are user-specified functions that are applied to specified object values to map them onto CLUT indices. Examples are conditional expressions that determine what value to map a particular object value onto. In Lisp this is straightforward. In C and C++, it involves a run-time interpreter which should be of as minimal complexity as possible. Other value methods deal with transparency, blinking, and logical operations on bit-planes.

2.2.2 Position Methods

Position methods are used to specify mappings from spatial objects and images onto two-dimensional display windows. They specify where to display something in a display window. For example, position methods are used to specify pan, zoom, and scaling parameters to relate an image to a display window.

The position methods and transformation networks used by the interface are defined by the coordinate transforms used in the IUE. The interface generally requires transforms of images and image-registered objects onto display windows and simple types of interpolation and sampling. More complicated mappings, such as image warping and generating rendered objects for a specific sensor, use methods from the sensor and scene classes and image packages for warping. These are used either to generate an image that is then displayed or to invoke the object specific display methods through the interface.

The coordinate transformations and networks are very important for the interaction methods. In this case, the user indicates some position in the display window, and then the mapping from the object to the display window is inverted so that the corresponding values and position in the object can be determined and accessed. The object display is able to do this for images and invertible geometric transformations. For others, such as interacting with a potentially complex, rendered solid, methods from the other classes, such as the sensor and scene classes, are needed. Possibly there are other representations of a rendered scene, such as an image-registered depth map, that contain pointers to all the surfaces that project to a given pixel, ordered by depth that can be generated to simplify the interactive processing.

There is often hardware-supported pan and zoom images that should be accessible through the position methods, even though this is machine dependent.

2.2.3 Interaction Methods

The **Interaction Methods** are for interacting with and inspecting objects through the context associated with a display, such as the current object-to-display transformation. The methods associated with this are built on top of the event-handling mechanisms of the supporting environment. Interacting with an object through a display involves using the position mapping from the object to the window. This is straightforward if the mapping is invertible and there is no interpolation or warping, which is usually the case for images and image-registered objects. It can also involve geometric intersection using the ray of projection corresponding to a selected window position. For other objects, such as closed analytical surfaces, the reverse mapping is more complicated and involves general spatial-object methods that need to be accessed by the interaction methods.

The current object to be interacted with can be explicitly specified or selected. Selection can require disambiguation if there are multiple overlapping objects or complex spatial objects. The user may be required to use a spatial index image (an image of pointers to objects which occupy a given position) or use geometrical data base operations in the IUE. Both are potentially expensive and don't reflect operations specific to the interface. They are general IUE spatial data base operations that need to be invoked through the interface to return the selected object(s) and object position from the object to display mapping.

There can be a variety of interaction devices (minimally it should specify a screen location), but we are assuming a mouse with at least two distinguished buttons and text-input from the keyboard. In the interactive mode:

- The current position of the mouse is stored in variables for the (*mouse-x,mouse-y*) of the current display window. Associated with this is the corresponding positions (*current-position*) and values (*current-values*) in the specified objects. The default involves only geometry to inverting coordinate system transforms and not sampling or interpolation relative to the objects. Since several objects can be interacted with at the same time, these lists will consists of lists of positions and values.

- When the mouse is clicked, the values for the window position, and the corresponding object positions and values, are stored in separate lists. Interactively selected Functions can then use items in these lists as parameters.

- Because a user can associate functions and command-sequences with keys and mouse-states, the functions can be called interactively. The functions are stored in a table index by mouse-state or keyboard-event. The functions can be a sequence of specified interface commands and can be interactively applied to the values in the different lists. Functions are selected using either the keyboard input (numbers) or mouse-state (mouse-down, mouse-up, mouse-drag for the left, right, and, if it is available, the middle mouse button). Function selection may also be based upon a count of the number of mouse-clicks for a specified mouse button.

- There may be a default spatial index associated with a display window. This is memory intensive but can help with a lot of the interactive operations, especially the selection operations.

2.2.4 Graphics

Often a user will want to perform graphic displays of text, two-dimensional graphics, and three-dimensional graphics. Examples are annotating a display, indicating where some action is occurring (the position of an epipolar line, translational flow paths, etc.), projecting a wire-frame of a model onto an image. Much of this functionality will come directly from an existing graphics package that the IUE will utilize. The graphic displays need to take place in three different modes. They can occur:

- in the coordinate system of the display window. In this case, displays only occur with respect to the window coordinate system

- in the coordinate system of a displayed object, such as drawing a line with respect to the inverse mapping from window-to-object coordinates

- in the corresponding IUE objects. Thus, in drawing a line in image coordinates, a corresponding instance of an IUE line object would be produced. When the wire-frame model is displayed, each line-segment and junction would be created as an object in the IUE. This is

very useful for producing data for testing routines. This mode can be coupled with the interactive processing mode to create data. This may be restricted to relatively simple objects, such as polygons, curves, etc.

2.2.5 Types of Object Displays

Four types of object displays stand out:

- The **image or pixel** display is for viewing images and image-registered features.
- The **local graphics** display is for displaying objects by mapping their values onto parameterized graphic objects, such as lines and cubes. Examples are displaying vector fields and edges.
- The **surface** display is for displaying objects that get mapped onto mesh or rendered surfaces.
- The **plot** display is for displaying functional relations between objects. Examples are one-dimensional, two-dimensional, three-dimensional graphs, histograms, scatter grams, and views of functions and tables.

Although, these different types are distinguished by specific methods, all inherit a large number of similar methods from the general display class. For example, overlay operations are similar for a surface display and for an image display, although they can look quite different. (In one case it appears like drawing in solid colors in image-registered coordinates on top of a displayed image, and in the other it would be rendering the colors onto a displayed surface.)

Local Graphics

Local Graphic Displays are a subclass of object displays that map object values onto parameterized graphics, such as a line, a square, a perspective view of a cube, Chernoff faces, or a user-specified function. A common example is a vector display that will map each component from a pair of images onto the x and y components of a vector. Using the general display methods, the vectors can be displayed as an overlay on top of an image or through indices in a CLUT. For visualizing three-dimensional attributes registered across an image, the user can display unit cubes with their orientation computed from the specified components of the display object. The graphic display can be a piece of graphics code, which will be positioned to the projected location of the pixel.

There will be specialized local graphic displays for vectors and different types of edges because of their heavy usage. It will be possible to display the horizontal and vertical edges in the cracks between pixels or to place a single edge at the center of a pixel with it's orientation determined by the specified components objects.

Plot Displays

There are several different types of plot displays: one-dimensional, two-dimensional, three-dimensional graphs, histograms, scattergrams, perspective views of functions and others. Examples of plot displays can be found in several data visualization packages and mathematics packages, such as Mathematica and GNUPlot. In using such packages in the IUE, remember the cost limitations on bundled software and the potential problems with data compatibility and speed. Also, plots need to be compatible with the general display methods for such things as the following:

- Mouse interaction methods: for selecting a position in a graph and then having access to the domain point and the range point. An example is interactive segmentation from a histogram displayed as a plot.

- Links between plot displays and other types of displays.

- Most of the view transformations for such things as scaling and zooming.

- Overlaying plots in different colors.

The plotting capabilities in GNUPlot is currently being explored to determine its use in the IUE. It is essentially free and all the source code is available.

2.3 Browsers

Browsers are used for interacting with text-based or symbolic descriptions of objects. They are used for actions such as querying sets of objects, determining and inspecting relationships between objects, process monitoring, and inspecting values in an object. The browser and browser-related classes are being designed so they can readily be built on top of existing interface construction kits.

There are two general types of browsers: **Field-Browsers** and **Displayed-Graph-Browsers** of which only field browsers are currently being implemented. Field browsers are built from component objects which are found in several GUIs.

- A **field** appears as a rectangular box which can be filled with text, icons, colors, colored text, text in particular fonts, or user-specified graphics. Fields can have actions associated with them when they are selected or when a user changes the values in them.

- Fields can be organized into connected **horizontal or vertical field groups**, where each field is a unique index in the field group. The fields in field groups will generally have different objects displayed in them. An example comes from the object-registered browser where a field group can correspond to a display of registered values from different objects. For better visualization, these can be displayed in different colors, fonts, etc., in addition to their position in the field group. A field group can also have a distinct boundary.

- Field Groups can be organized into field matrices where each group is a unique index set in the field matrix. Objects and sets of objects can be mapped onto the matrix.

- Field Matrices can be scrollable as a way to control the mapping of an object (or object set) onto the field matrix.

There is a distinction between four types of field browsers, which are inherited from the general Field browser class.

- **Object-Registered Browser**: This contains values extracted from a spatial object, such as the intensity values in some square neighborhood of an image. Depending on the dimensionality of the object (or relationships between component objects), this can be presented as a one-dimensional array, a two-dimensional array, or multiple two-dimensional arrays, and can be used to describe curves, images, image sequences, and pyramids.

- **Set / Database Browser**: This is presented as an array of fields. Each row of fields corresponds to selected attributes of a particular object, and each column corresponds to

common attributes over the set (or database) of objects. An example would be browsing the database that describes the current active objects in the IUE to find the most recently created image from some operations.

- **Object Attribute Browser**: Each row corresponds to the value of an attribute for an object. This would be used for inspecting a single object.

- **Graph Browser**: This browser is useful for text-based inspection of graph structures and trees. When an item is selected, the related items (along some relational dimension) are displayed in the next column.

The methods associated with browsers are very similar to those with displays, suggesting a more general IUE interface object class. The position methods for browsers involve how an object (or set of objects) gets mapped onto the fields of a browser. Mapping object-registered browsers, is essentially the same as displays (see below). The fields are analogous to pixels in a display window, although they can be filled with textual information. For DataBase browsers, the position methods specify how objects are mapped onto rows of the browser and how attributes are mapped onto columns (see below). The position methods for mapping from graphs and networks onto a Graph browser involve keeping track of different paths through networks and nodes and arcs that have been traversed. Browsers can also be linked to browsers, displays, and user specified interface widgets.

The following examples have been implemented using the FORMS GUI kit on SGIs.

2.3.1 Object-Registered Browser

The Object-Registered Browser is used to inspect the values in a neighborhood of a spatial object. A common example is inspecting the image values about a selected point. It is similar to displaying a spatial object in a display window, but instead of the values being mapped onto window positions and screen intensities and colors, values are mapped onto field locations and general field attributes, such as colored text in specified fonts, colors, and icons. The attributes and the specification of the mapping from an object onto an Object-Registered Browser is shown in Figure 2.8. A set of spatial objects are mapped onto a matrix of fields in the Object-Registered Browser. This mapping involves several parts: a coordinate transform from the N-dimensional spatial objects to the two-dimensional Object-Registered Browser; the type of interpolation to be performed if this mapping doesn't involve discrete values; what to do when browsing beyond the boundaries of the spatial object. Also shown is a navigation tool to interactively access position methods to position the Object-Registered Browser with respect to a set of spatial objects. The browser is linked to a display window that shows the position of the browser with respect to the bounding rectangular prism of the spatial object. This display would be updated when the browser is moved with respect to the spatial objects.

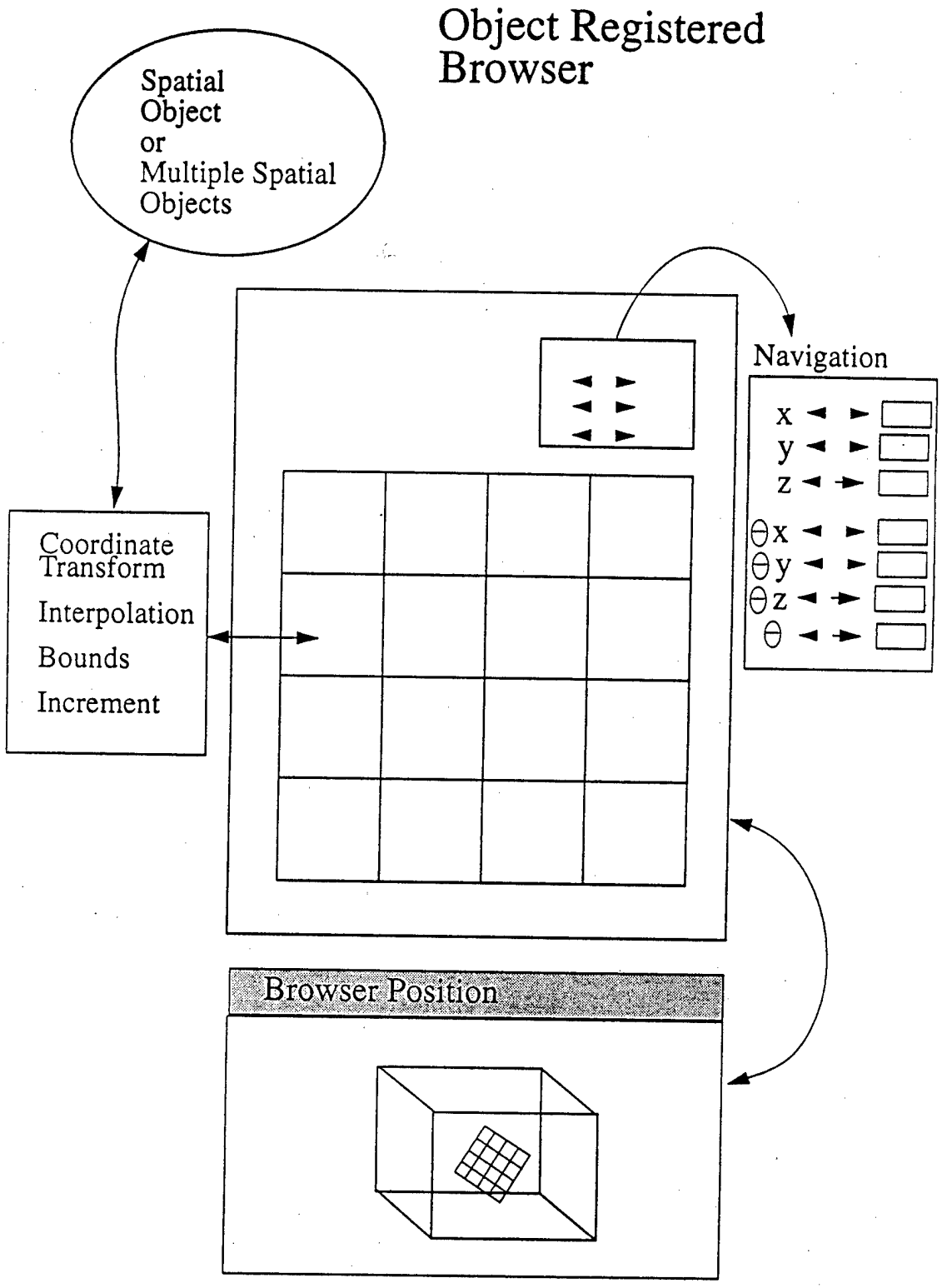

Figure 2.8. Object-Registered Browser.

Figures 2.9 and 2.10 show an implemented two-dimensional Object-Registered Browser. In Figure 2.9, a single image is displayed with values mapped onto intensities, text, and an icon.

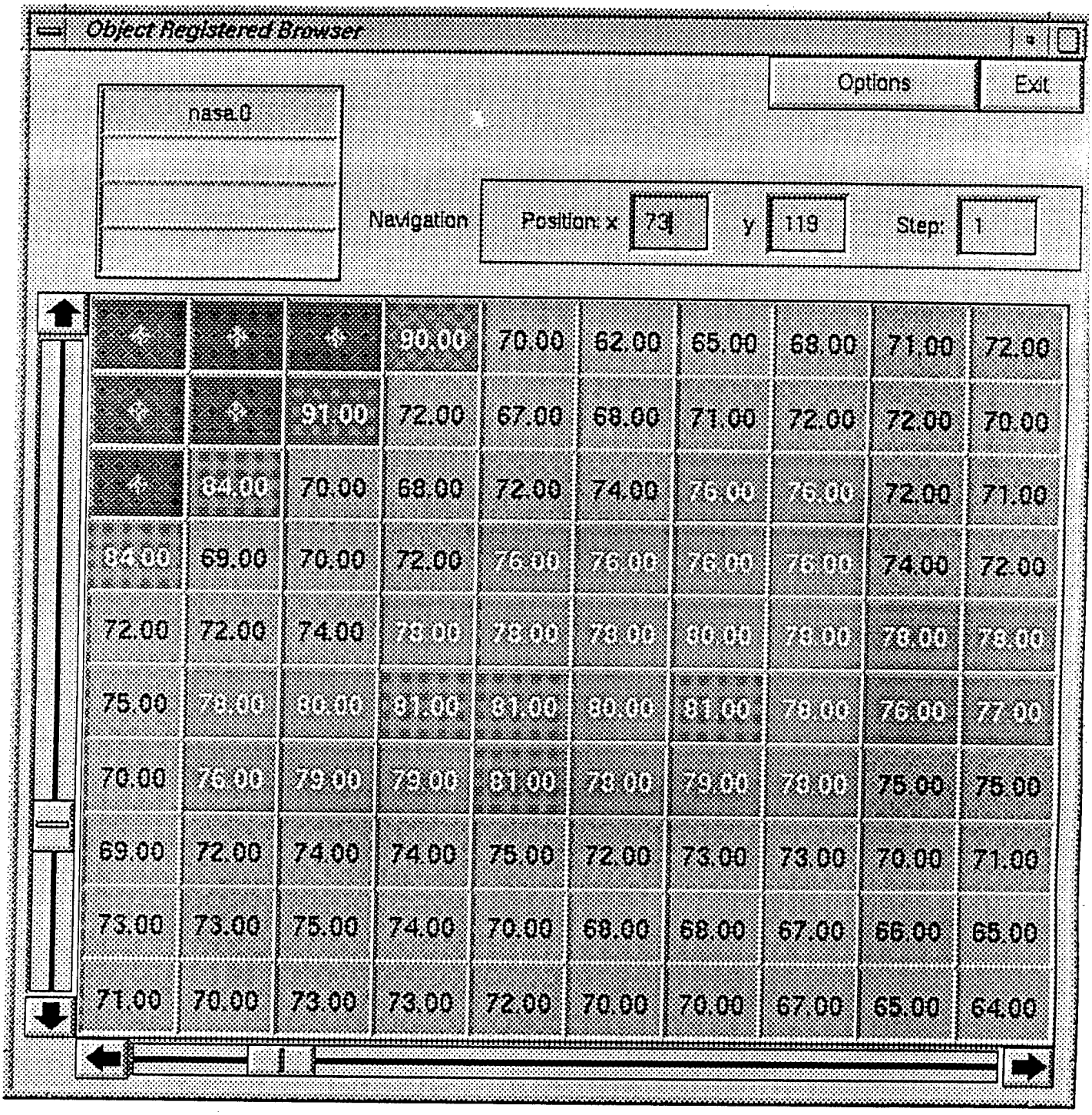

Figure 2.9. Object-Registered Browser applied to an image.

Figure 2.10 shows two images and the computed difference of the two images, each in separate fields. Each image is displayed in a different color, and the field containing the difference image uses the background color to encode the magnitude of the difference.

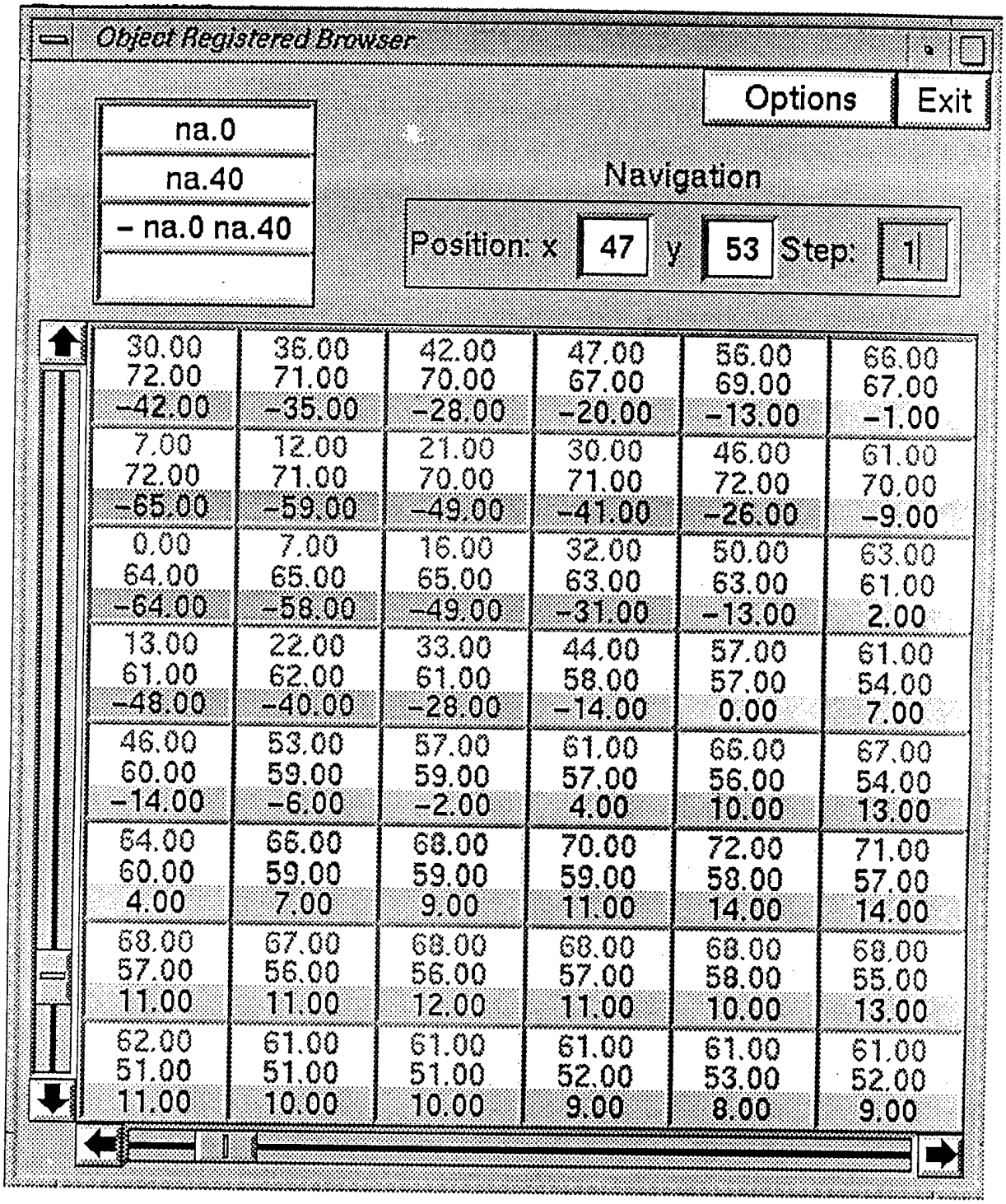

Figure 2.10. Object-Registered Browser applied to an image.

2.3.2 Set/Database Browser

The Set/Database Browser is for inspecting the attributes of a set of objects, which enables interactive queries to be performed via the browser. This is especially useful for keeping track of instances of objects (an object selected in a Set/Database Browser should probably default to the *current-object* so it could be displayed immediately). There are two structures used for describing the mapping from a database onto the browser. One is the set of selected attributes which correspond to the columns. The other is the current set of items which satisfy a query and the indices into the first and last element of this set, which are displayed in the corresponding rows of the browser (see Figure 2.11).

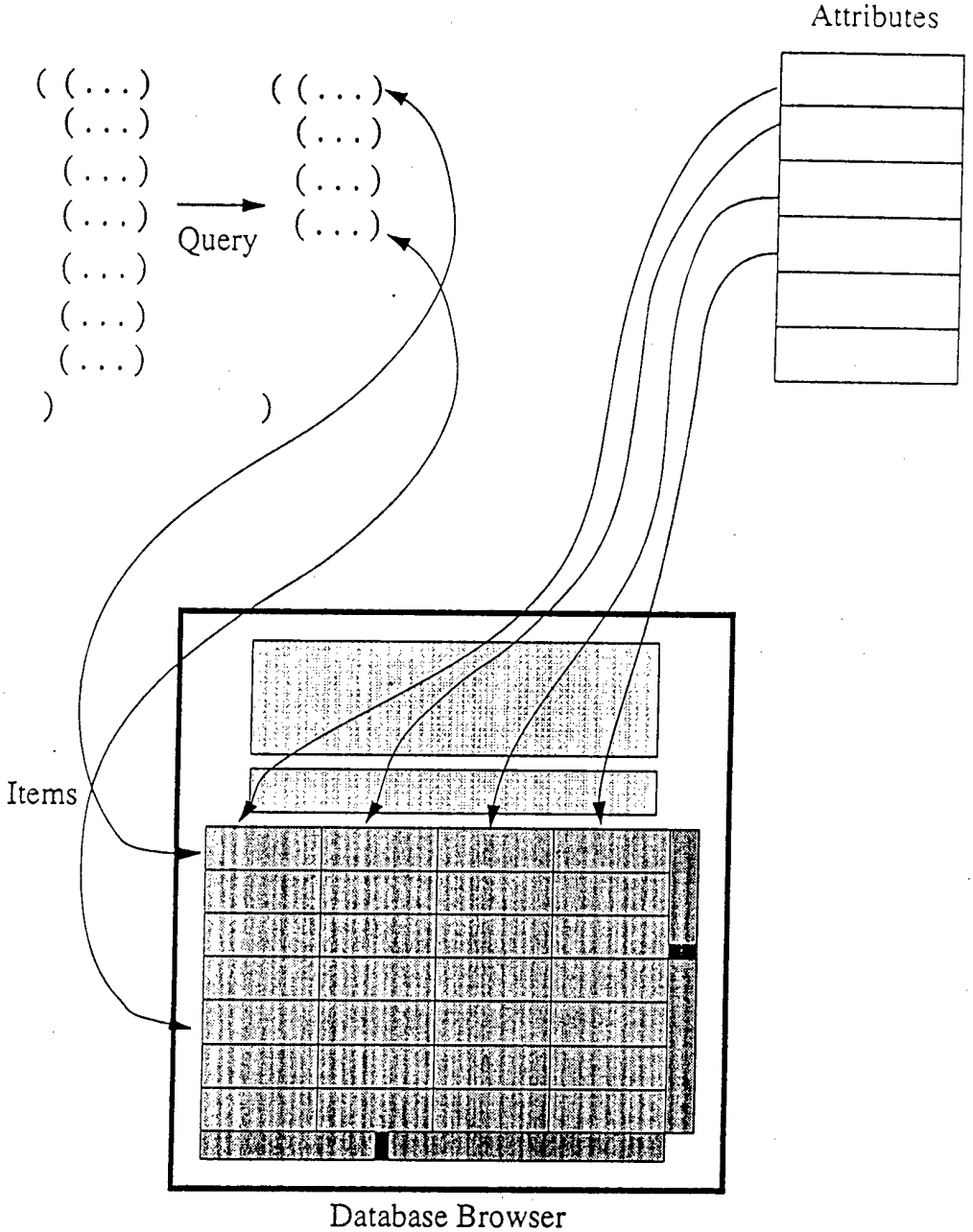

Figure 2.11. Set/Database browser.

Figure 2.12, shows an example using the Set/Database browser to inspect a set of line segments and then to sort them by slope

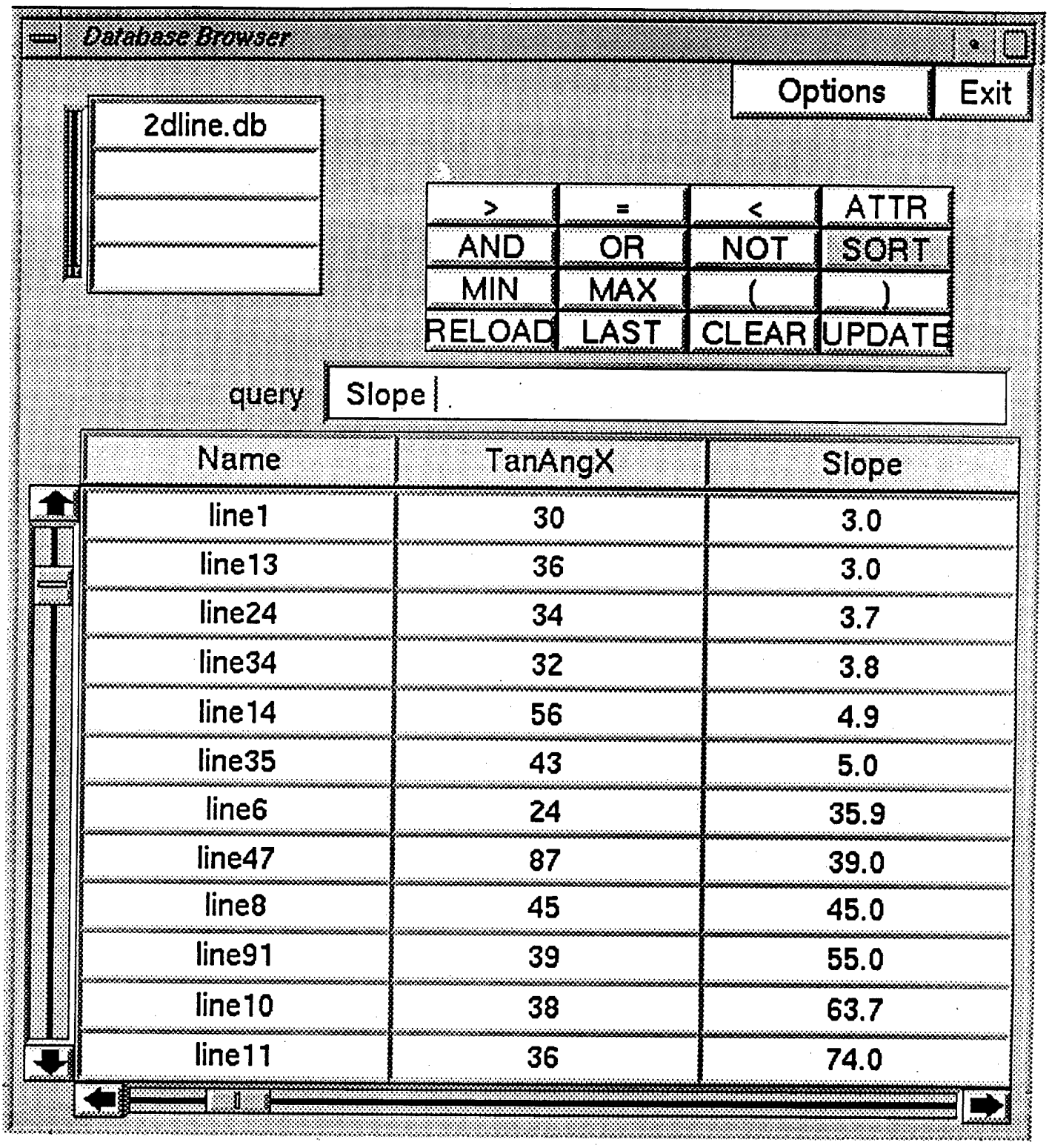

Figure 2.12. Set/DataBase browser applied to a set of line segments from Data Exchange Format.

2.3.3 Object Attribute Browser

An **Object Attribute Browser** is for inspecting the attributes of an object or several objects with the same types of attributes. It uses an ordered list of object attributes to determine which attributes of the object to display. Figure 2.13 shows an example of a Object Attribute Browser applied to the attributes of an image object.

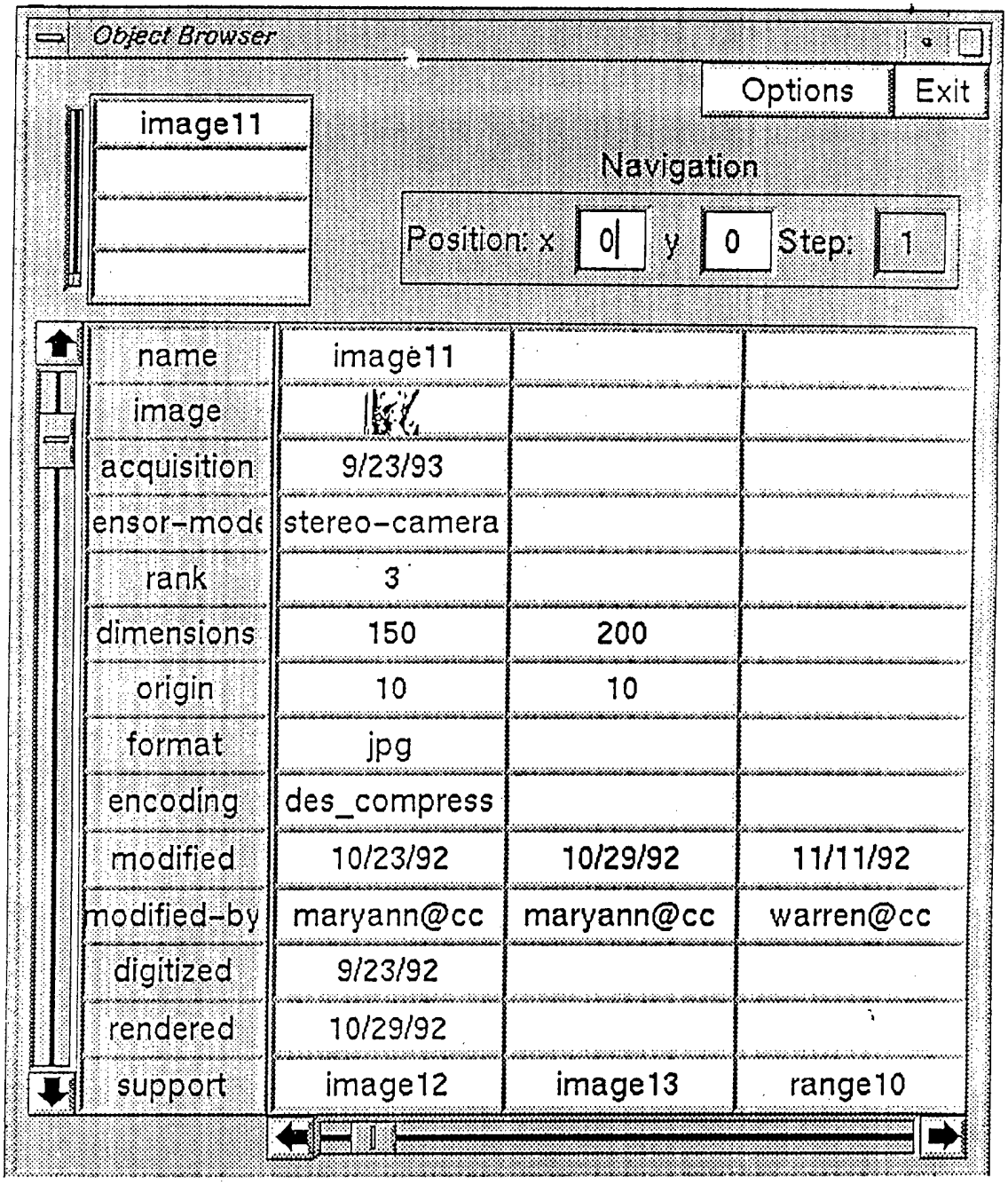

Figure 2.13. Object Attribute Browser applied to an image.

2.3.4 Graph Browser

A **Graph browser** is for inspecting graphs and network objects. Instead of one large field matrix, it consists of linked Nx1 field matrices. Each column corresponds to a set of nodes. When a node is selected, the types of relations (arcs) are displayed in the *current - arc - browser.* When a type of arc is selected, the nodes with that type of relation are displayed in the adjacent (right) column. Several structures are used to describe (and update) the mapping from the graph onto the successive browser columns. The current node is the most recently selected node. The current path is stored, as well as the nodes that have been visited. Figure 2.14 shows how a network is mapped onto the graph browser. Figure 2.15 shows a Graph browser applied to a polyhedral mesh.

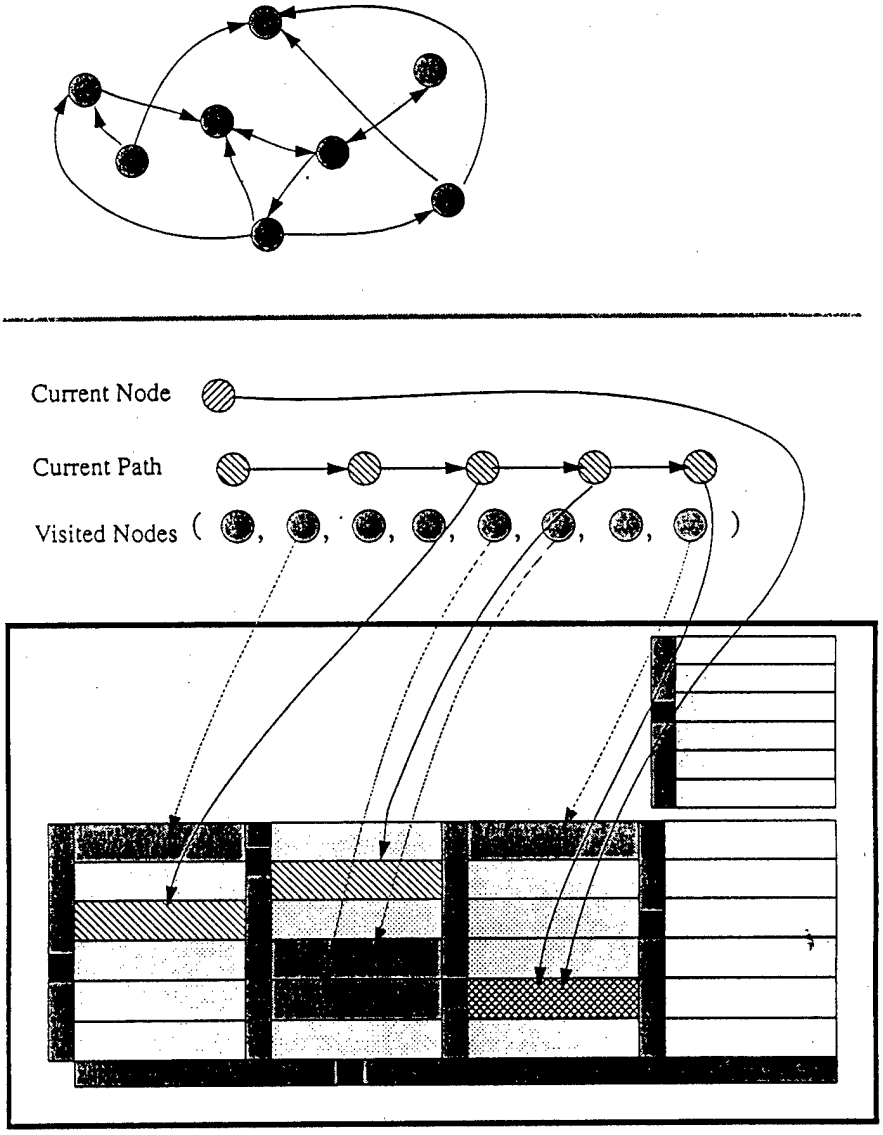

Figure 2.14. Mapping a network onto a graph browser.

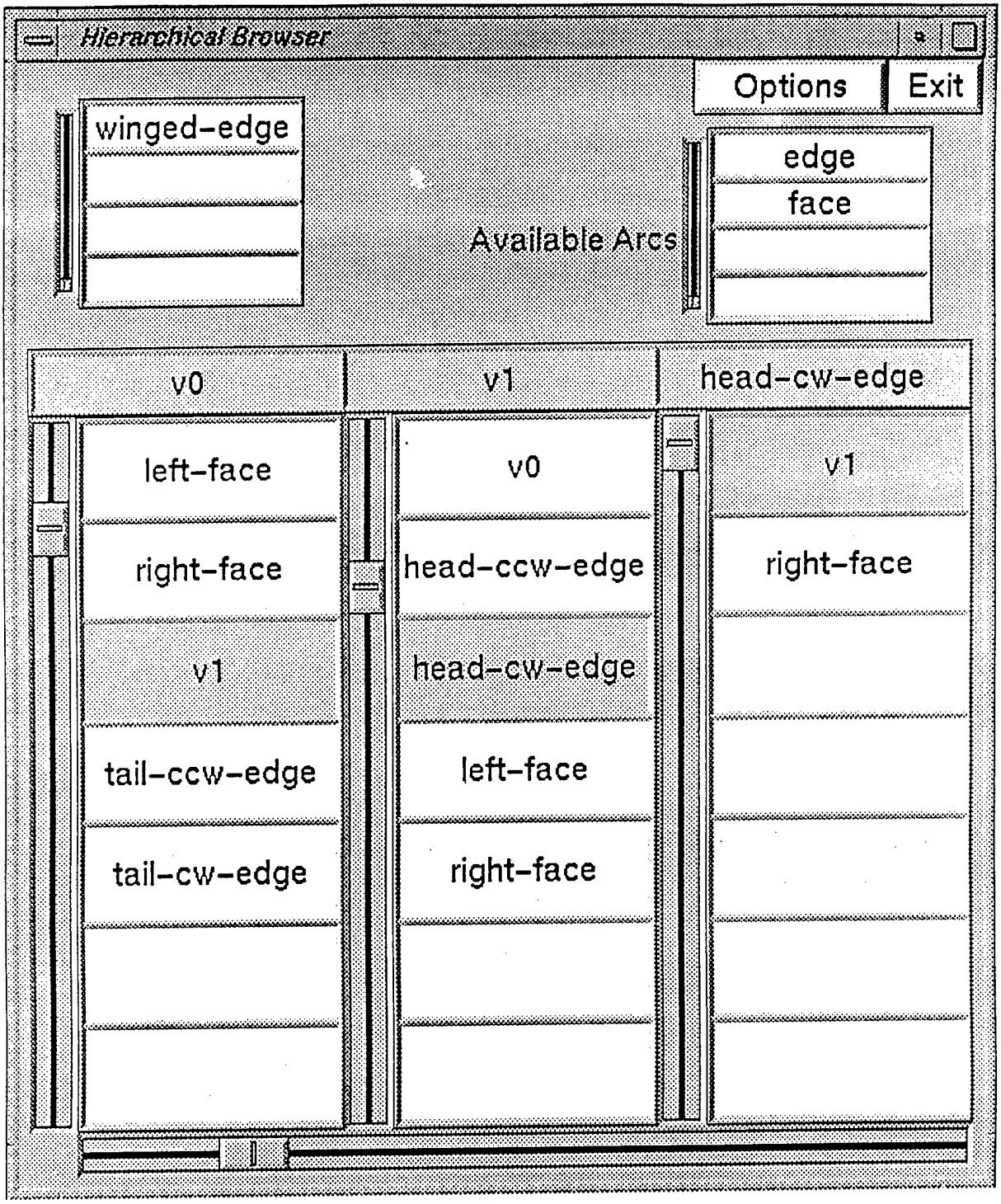

Figure 2.15. Graph Browser applied to a description of a polyhedral mesh from the Data Exchange Format.

2.4 Interface Context

There are several data structures for describing the context of the interface. These are used for intelligent defaulting and for saving the state of the interface. They include:

- **Object-Display Mapping**: Structures that describe the mapping from an object onto a display. This includes viewing the transformation between an object and a display window, the value-mapping of how the object is displayed, and a reference to a particular CLUT.
- **Object-Browsing Mapping**: Structures that describe the mapping from an object(s) or database onto a browser.
- **Display Context**: Structures that describe the current context for a display for such things as the current window, the current object, the current object display mapping, the current display command, the current mouse-selected object position and value, and the lists of interactively selected object values and positions. For example, if neither a display nor an object is specified, it will default to the most recently used.
- **Browse Context**: Related structures for browsers. Such things as the current browser, the current data base, query history, and others.
- **History**: The sequence of display or browsing actions for a particular window or browser are saved and can be re-accessed and used for creating animations. In addition, objects that have been displayed or browsers are also stored.
- **Default layouts for windows and browsers**: The desired layout of windows and browsers can be saved and can be available to a user when starting the IUE. Users may prefer different interfaces (arrangement and instantiation of the basic interface objects) depending on the task or level of sophistication.
- **Object Display Links**: A structure that describes the concatenation of a display or browsing operation between IUE interface objects.

The context description is an extension to the underlying context usually provided by the graphics level. It should be possible to read and save context descriptions.

2.4.1 Links

Links support operations such as window to window zooming, displaying the same object from different views or using different value-mappings and controlling displays using interactive widgets like sliders and knobs. Linked displays are useful for displaying composite data such as stereo image pairs or pyramids. When something happens in a parent display (or browser), another display will perform an action using information from the parent display. The action can be a display operation or a sequence of commands associated with the link, such as a set of commands from the interactive command language.

The mapping between a spatial object and a display in one window can be concatenated with the display specification in another. A common example is using one window to zoom onto the display in another, or using one window to display a selected portion of another (panning and zooming are so common they will be directly supported via an interactive tool).

There are specified constraints on links to avoid many complex and events. Linked displays and browsers are only updated when a display action is performed, not when changes are made to the displayed object. Individual links are bi-directional, but no cycles are allowed in the graph formed from all of the links between IUE interface objects.

2.5 Command Language

Users will be able to specify all interface actions through an interactive command language and will be able to access all the functionality of the interface. Display operations can be performed interactively through the command buffer. The command language will have intelligent defaults and abbreviations (such as displaying to the current window if none is specified). In addition, the commands will be used in non-interactive code for creating scripts and general display routines. All of the functionality of the interface is accessible through an interactive command language, which encompasses the overall functionality of the interface.

A concern with the interface command language is that it becomes another language that people will need to memorize. This is not an issue for development in Lisp since the display operations can be called interactively like any other function, but it is a significant issue with C++. The command language is intended to be as simple as possible, with a limited syntax. Most arguments are specified via keywords and correspond directly to interface object methods. There are also defaults for command specification. The IUE will eventually support intelligent prompting to complete the commands.

The general syntax is

```
IUE-interface-object  object-set  [keyword  arguments]*
```

For example,

```
[*w1*  image1  :p]
```

means to display image1 using a pixel-type display in window *w1* using the current display context associated with the display in window *w1*. The brackets are used to indicate separate commands. If the last display operation was of type :p in *w1*, then only:

```
[image1]
```

needs to be specified. More detailed examples are presented below.

An important operation for displaying spatial objects is the ability to apply functions to objects prior to displaying or interacting with them. These operations almost always don't involve creating a new object. An example is manipulating the underlying CLUT to perform a thresholding operation. In this case, no thresholded image object is produced, only what is displayed in a window is produced. This goes by many names in different systems, such as Pixel Mapping Functions, Dynamic Color, Generalized Color Look-Up Tables.

There are two aspects to such functions. First, there are limitations on the types of functions that should be specified for application to an object when it is displayed. Operations, such as zooming, panning, manipulating the CLUT, specifying which planes in the display buffer are used, and using simple point-wise algebra with limited conditional evaluation, are very useful and will be supported. But, it doesn't make sense for operations, such as generalized warping or detailed

processing over a neighborhood or generalized intersection, to be done by via interface commands. Second, there are also language specific aspects for specifying function application to objects prior to display. In Lisp, it is straightforward to pass lambda expression or closures which are applied to each position or value prior to display. In C, this requires a library of standard functions and an interpreter.

In the actual operation of the IUE, it is not necessary that all interactions take place through this command language: some will be invoked by menus and special keys, and refer to the current display context. An important part of the design of the IUE interface entails how commands (and which commands) are mapped onto menus and other interactive devices. This is especially important since the interface will support a wide community ranging from naive users, who are interacting with hardened applications, to developers. Naive users may want many interactive devices such as specialized menus, while experienced users will want more powerful, abbreviated commands. Advanced users will become very adept at the shortcuts that should be provided.

2.5.1 Examples

The following presents some examples of specified display operations using the command language.

```
[*w1*  image1  :p  :linear  0  128  *screen-min*  *screen-max*]
```

This would display to window *w1* using the current defaults. The range of object values from 0 to 128 are linearly mapped onto the range of values *screen-min* and *screen-max*.

```
[image  :p]
[edge-image  :overlay  red]
```

An image is displayed in the current window using a pixel-type display. The edge image is then overlayed on top of this. Wherever the edge-image is equal to 0 nothing is displayed in the red overlay plane, and wherever the edge-image is equal to 1, the corresponding pixel is set in the red overlay plane.

```
[:overlay-colors  (red,  green,  blue,  violet)]
[image  :p  :value-function
              (if  (image.value  >  10)  red blue)]
```

The first command tells the current display to use the specified overlay colors. The second will display red in the overlay plane at a screen pixel corresponding to an image pixel if the image value is greater than 10, otherwise it will display blue.

```
[spatialIndexImage
   :p
    :value-function
      (if  (label-image.value  =  NULL)
         0
            (length  (spatialIndexImage.value)))
    :linear  0  20  0  *screen-max*]
```

This function displays a spatial index image (an image where each pixel contains a list of all the objects which occupy that pixel). The value function determines the number of objects in this list and the linear function maps this onto available screen intensities.

```
[image1  image2
      :p
       :value-function
          (image1.val  -  image2.val)
       :linear  -20  20  *min*  *max*]
```

The above code will display the difference between two images. Other common value functions would be for type conversion and display histogram transforms. The user can also specify functions in the interactive mode to be applied to the values in the different queues. For example:

```
[image
    :i
     :1  [p  :overlay-plane  clear]
         [p  image  :value-function
                    (if  (image.value  >
        object-values[1])
                         red  blue)]
```

The user has selected an image location with a mouse click and the corresponding queues have been filled with the window and object positions and values. Thereafter, when the user hits the terminal key 1, the overlay planes will be cleared and all image locations with a value greater than the value at the selected image location will be displayed in red, otherwise blue, in the overlay planes. *Image.value* is a dummy variable that refers to the current value in the image that is being displayed. *Object-values[1]* refers to the value selected using a mouse click in the display window and stored in the object-value queue. *Red blue* refers to globally defined overlay colors. Recall that the *:value function* specifies the operation to be applied to an object value to map it onto a screen intensity or color.

```
[:link  *w1*  :zoom  2  2  :pan  50  50]
```

This links *w1* to the current window and concatenates a zoom and a pan transformation.

```
[RegionDB
    :p
     :positions  RegionDB.locations
     :values  RegionDB.textureMeasure
    :linear  0  100  *min*  *max*
    :red-8]
```

This says to display the RegionDB in the current display window with the positions coming from the locations attribute of the regions in the RegionDB and the values by taking the RegionDB texture mappings and using a linear mapping from these onto screen intensities in 8 bits of red.

```
[*W1*  histogram  :plot1d  ]
[*W2*  image  :p]
[*W1*  histogram
    :i
     :1  [min  =  object-values[1].x]
         [max  =  object-values[2].x]
        [*W2*  image
            :p
             :value-function
                (if  ((image.value  >  min)  &&
                      (image.value  <  max))
                   blue  red)]
```

This is an example of plotting used for interactive histogram segmentation in which the interaction methods let us click on the axis of a plotted function to return the x coordinate and the y-value of the displayed object and then use these values to specify peaks in a histogram. Here the user has plotted a histogram in *W1*, and then selects the range of values by clicking on the displayed histogram. The current-object-value contains the x and y value from the displayed histogram. These are stored in the local values min and max. When the user hits the key 1, the selected range of values are displayed in the blue overlay plane in *W2*.

2.6 Additional Features

Even though the focus has been on developing the core functionality of the user interface, there are several other features that have been considered for use with the interface. Some of these can be built on top of the interface objects and operations described previously. These are important candidates as packages and libraries to augment the core IUE.

One important area involves interactive task management tools. Examples can be found in the data-flow editor in the Cantata portion of Khoros and the Task editor in KBVision. Another area that is important in developing graph browsers for the display of graphs and networks is to represent objects or values as nodes and to use links to describe relations. Graph browsers can have difficulties when trying to display several nodes with arbitrary relations between them in that the connections between the nodes can begin to obscure the overall display. A typical use would be to display a constraint or coordinate transform network.

There are probably hundreds of nice interactive controls for displays and visualizations that exist in different environments, such as interactively manipulating the object-value to screen-intensity function by shaping a function; selecting CLUT; modifying CLUT; interactively building display commands using templates or command browsers; floating tool palettes of interactive drawing tools; etc. In general, such tools can be very useful, but it is extremely important that there be a consistent look and feel with different applications that are based on the IUE. This will be partially achieved by depending on the underlying graphical user interface to supply the basic interface objects.

Other useful interface tools are:

- Interactive Selection and Modification of CLUTs and display mapping functions; cycling through different CLUTs.
- A dialog box for setting up system defaults and initializing characteristics of the IUE: initial layout, font selected, level of expertise, etc.
- Access to an integrated use of Established Visualization Packages: There are several data visualization products and it would be nice to have a modular interface to these.
- Mensuration tools: Such things as rulers, grid overlays, and the use of multiple cursors as a mark of distances and points of reference. These probably can be built on top of basic interface capabilities, displaying IUE objects (in particular, the display interaction methods and IUE objects, such as bit-mapped regions and line-objects).
- Interactive Object Creation (Draw Objects): It should be possible to create objects interactively. This is useful for creating idealized data for testing and development. It should be supported

by the display interaction methods and by access to the instantiation methods associated with spatial objects.

- Incorporating Hardware Accelerators: So the interface and the IUE in general can modularly incorporate different hardware accelerators.

- Display Buffer Optimization: The display buffer itself is a short-term memory for manipulating the view of a displayed object. A useful feature would be routines to directly access the display buffer or perform specific display operations in ways optimized for particular types of displays.

Chapter 3

Hypermedia Annotation for Tutorials

3.1 Introduction

The IUE will be supported by on-line documentation and tutorials. The tools for implementing these will also be available for enhanced communication and publication by scientists and developers who use the IUE. While there is significant activity in developing documentation and hypermedia toolkits, they remain largely machine dependent with no clear standardization. A simple documentation tool called Knowledge Weasel (KW) is being developed. KW is based on Lucid Emacs 19 and existing media editing tools.

The organization and implementation of the Knowledge Weasel (KW) Hypermedia Annotation System, which is currently being used to explore knowledge structuring by collaborative annotation, such as a large group of people reading a book together and engaging in prolonged, asynchronous conversations is described in this chapter. KW incorporates many useful features: a common record format for representing annotations in different media that allows uniform access to them; dynamic user control of the presentation of annotations as a navigational aid; global navigation using queries and local navigation using link following; and support for collecting related sets of annotations into groups for contextual reference and communication. KW purposely leverages off of free, publicly available software so it doesn't require building specialized tools and can be freely available. Some issues discussed involve annotating non-textual material, such as images and sound, and conclude with a brief discussion of ongoing and future work.

Knowledge Weasel (KW) is a hypermedia presentation and authoring system designed to support collaborative annotation using several types of media. A simple analogy for KW is a group of people reading a book or attending a lecture and being able to make diverse types of comments and annotations on the material to supplement the material and make it easier to understand. Or consider the knowledge that is created during a course, in terms of understanding what develops in the minds of individual students that is usually discarded and not accessible to future students. KW is intended to extend the static information found in existing courses and books by developing a rich infrastructure to increase accessibility and different organization of material. It should be easier (or more rewarding) for the 5000th person who has worked through a textbook to read the textbook than the first person, in the same way that paths through forests become clearer over time.

In reality, such unrestricted annotations and comments, made with respect to real books and lectures by hundreds or even thousands of people, would create a significant mess. Therefore, in developing KW, tools were created to extend this simple metaphor in several ways:

- Using multiple types of media. There is support for several types of annotation, distinguished by media type (such as images, text, sound, drawings, and eventually including video and gestures) and their functional role as an annotation (a comment, a counter-example, a question, an analogy, etc.). For annotating non-textual media, such as images and sounds, a distinction

between two different types of annotation is introduced -- *superficial annotation* and *deep annotation*, each requiring different types of tools.

- Developing a general format for annotations. A common annotation record that is wrapped around different types of media for uniform accessibility is used. The annotation records are structured to make possible intelligent and automatic processing, eventually including rule-based processing for automatic presentation and "ferreting" of information (hence the name).

- The need for different types of navigation, organization and presentation tools to keep users from being overwhelmed with a great deal of possibly irrelevant information. KW supports two basic types of navigation. One type of navigation is done via queries over a database of annotations (*global navigation*). Another type is done by following local links between annotations (*local navigation*). The query-based navigation is generally used to search globally over a set of annotations to set up a context for more local link following. The sequence of queries, and the displayed sets of annotations to define a collection of related annotations, can be manipulated as a whole for reference or contextual communication.

- User control of annotation and button presentation. This is critical for users to perform navigation without getting lost, especially in the case of a heavily annotated document. The materials being annotated can be completely covered by the indicators (buttons) corresponding to the annotations. The display of links and buttons can be suggested by an author but is not fixed by him. Whether they are displayed, and their visual attributes, are dynamically controlled by a user.

- Supporting concurrency of access so multiple users can synchronously annotate a document. This has also involved experimenting with protocols for who can change what in annotated documents, and how to update documents based upon these changes.

- To avoid duplication and to be aligned with ongoing developments with the software world as a whole, KW is based on existing (and essentially free) file, database, and media editing tools. This includes operating systems and file management facilities, text editors, and X-based tools for dealing with different types of media, such as sound, images, and graphical user interface construction kits. Significant leveraging occurs because of this. For example, because of the integration with GNU tools, running programs can be annotated while they are being executed in a debugger without requiring the introduction of additional capabilities. It is suspected that the distinction between hypermedia and general operating system software and file systems will lessen in the future and hypermedia will become more and more a conventional part of computer system software.

KW has been influenced by, and shares attributes with, many major hypermedia systems, especially Notecards [10] and Intermedia [29]. Intermedia has been a major influence in terms of the use of hypermedia in instruction and InterNote [5], the extension to Intermedia for collaborative annotation, which stresses the importance of a common annotation format and the metaphor of making annotations on "layers of acetate" over different media.

Figures 3.1 through 3.4 are brief segments from sessions by different people using KW. The processing in Figure 3.1 involves an author who wrote a research paper and is using it as the basis of a tutorial for a class on computer vision. The author is going through his paper and annotating it with examples and with detailed descriptions of how concepts in the paper are reflected in the corresponding code. The author first selects various files of text, results, and code and then specifies that these are files to be annotated in KW. In Figure 3.1, the author is creating an annotation between a selected portion of the paper and the corresponding code. Shown is the annotation with the textually displayed buttons (they are actually in large blue text) and the corresponding areas of the text and code.

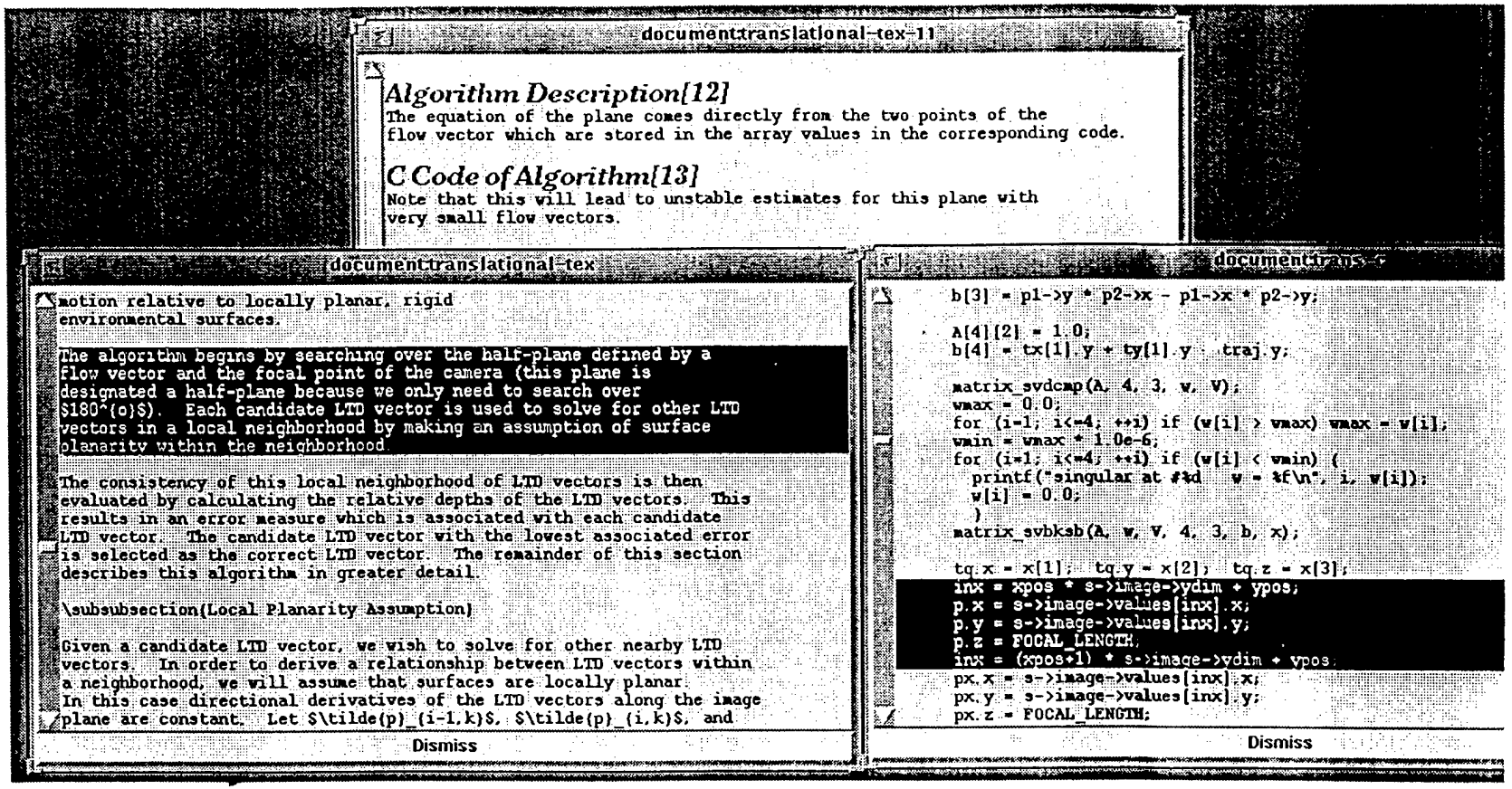

Figure 3.1. Author annotating text.

In Figure 3.2, a student begins to interact with the annotated paper and tutorial. The student is using the Database Browser to perform a sequence of interactive queries with respect to the set of annotated documents. The student has specified that he wants to see all annotations made by the author on the document and then ordered these by time. Annotations can be viewed by selecting them one at a time from the Database Browser. Or, the Database Mapper can be used to map the buttons corresponding to the selected annotations onto different visual attributes, such as size, font and color, and then interactively select the annotations.

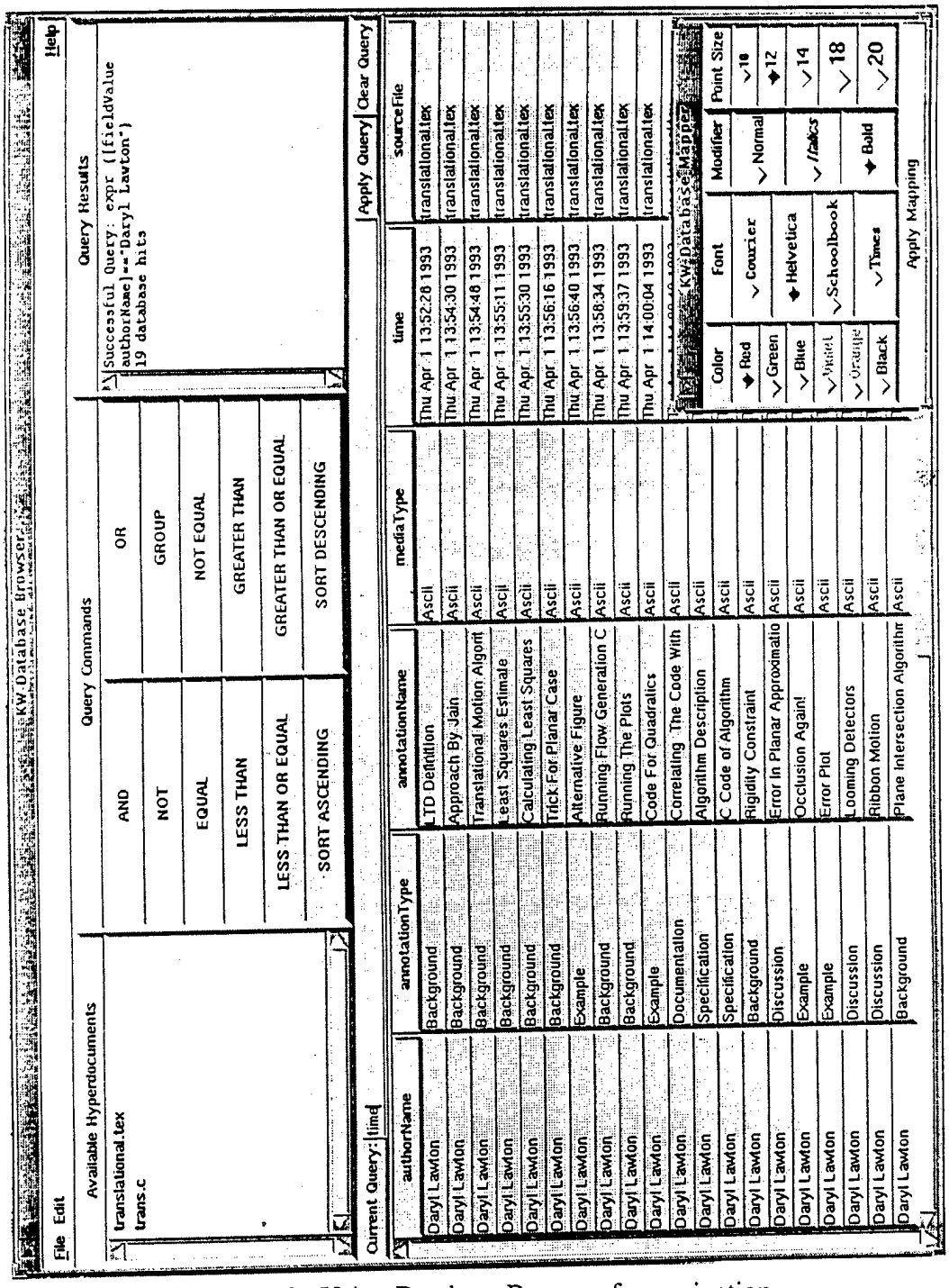

Figure 3.2. Using Database Browser for navigation.

In Figure 3.3, the student has been executing some code described in the text and found a strange result. First, the corresponding portion of the text document might be queried to see if any one else has found a problem or been confused by the materials in this area of the document. Not seeing one, an annotation is then created that links the code and the strange result (indicated by annotating the displayed image of the error function by marking the displayed image) and stores this with respect to the original document.

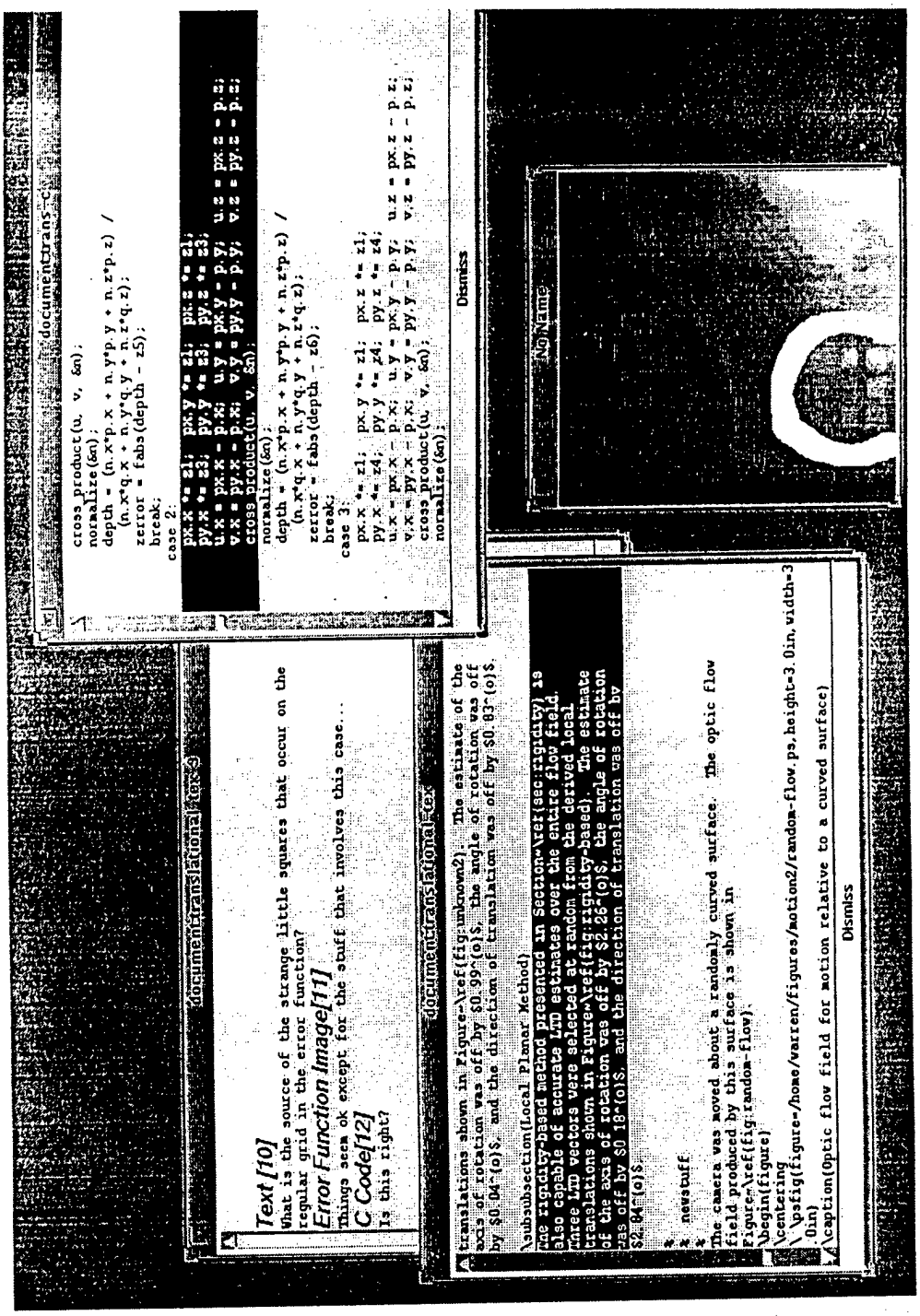

Figure 3.3. Questions expressed as annotation.

In Figure 3.4, the author looks over the paper. The document can be scanned for reactive comments made since a particular date. In the figure, the author has grabbed a potion of the text and asked for any comments that were questions on the corresponding portion of the text and are directly connected to such questions.

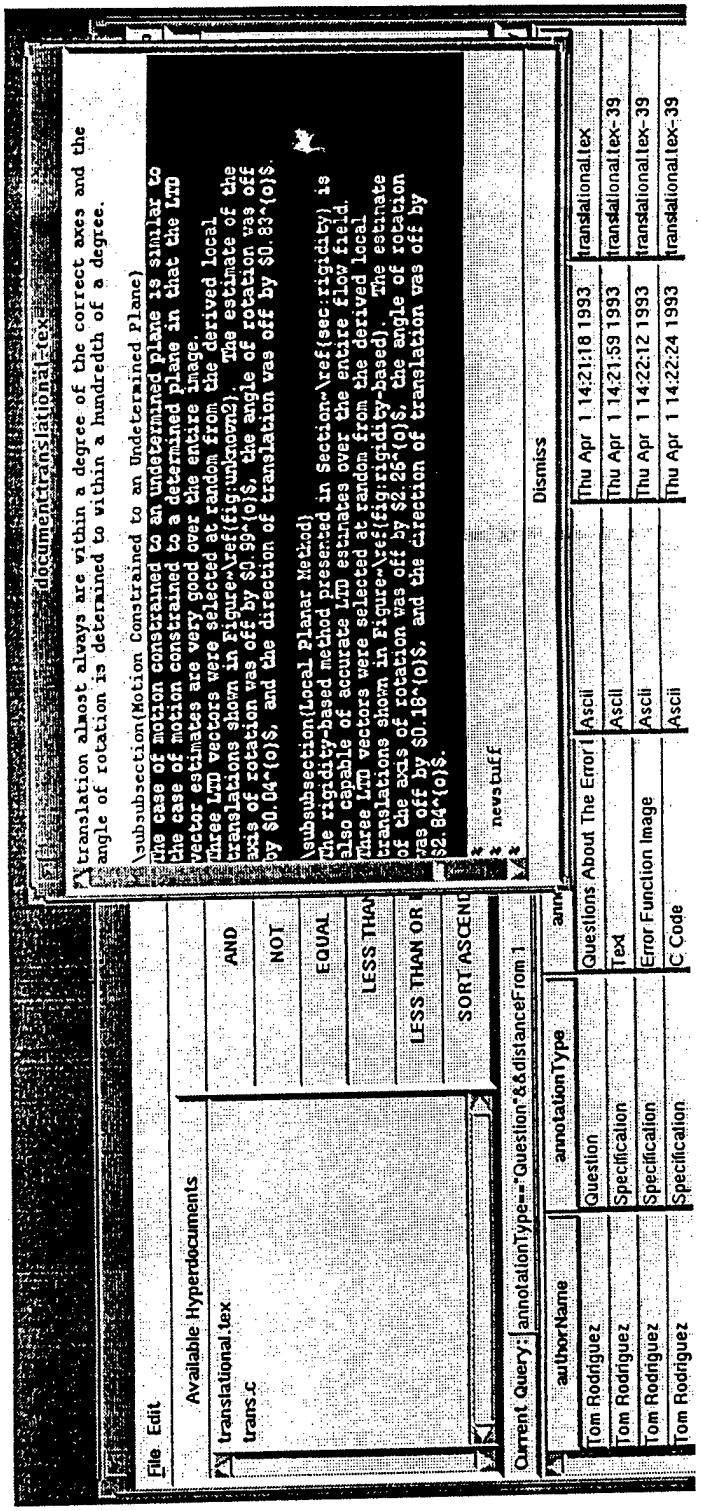

Figure 3.4. Author reviewing responses.

In section 3.2, the basic architecture of KW is described and other parts are discussed in more detail. Section 3.3 looks at the representation of annotations and issues in treating annotations involving different types of media in a uniform manner. The database aspects of KW and how queries can be used as a type of global navigation are explained in section 3.4. Section 3.5 discusses how the presentation of annotations is controlled by users. A description of how KW was implemented is presented in section 3.6. The conclusion in section 3.7 will discuss ongoing and future work.

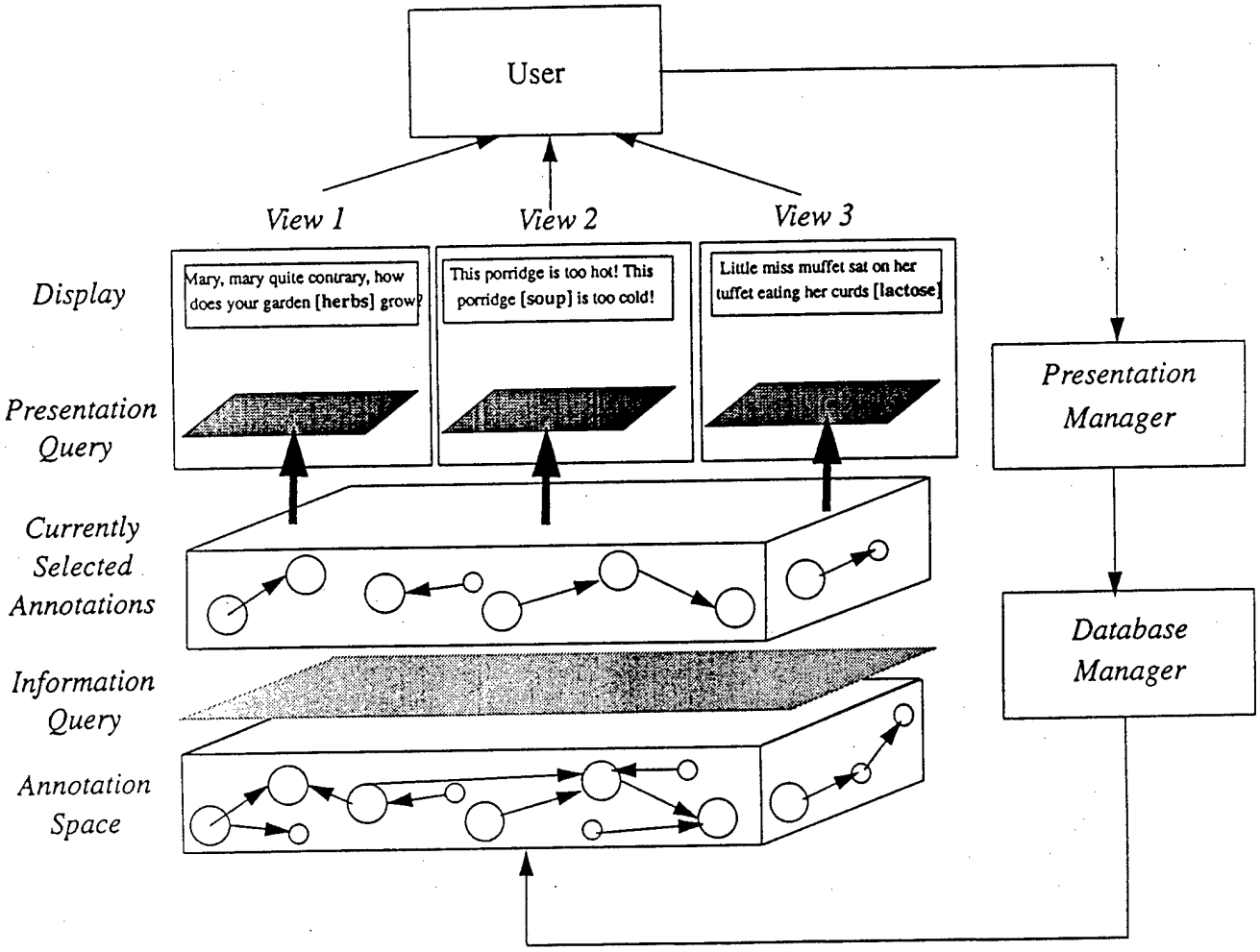

Figure 3.5. Knowledge Weasel Architecture.

3.2 KW Architecture

The logical architecture of KW is shown in figure 3.5. KW consists of different levels of data representation related by user-directed processing facilitated by different "managers." The lowest level is the *annotation space* that consists of all annotations. Users can query the *annotation manager* to select a set of annotations to interact with. Selected annotations are displayed by the *presentation manager*. This involves how to visually present the indicators (buttons), which correspond to annotations on a displayed document and also determines which button is selected using a mouse. The display of selected annotation is highly contextual and under user control. The *presentation manager* consists of several "annotation renderers," which display the selected annotation buttons in a manner appropriate for the different types of media. Selected and displayed annotations can be grouped together to form "views." Views supply a way of organizing and referring to different contexts for storage and communication. A user can have multiple simultaneous views that he is interacting with.

There is no real distinction between authoring and navigation in KW. It is important to navigate to the information that one wants to annotate, and it is important to take any type of material as something that can be annotated. Given a context or a selected document, a user can annotate it directly. The user can use available system media editing tools to create new data files that can be annotated.

A central idea in KW is that of an annotation (see Section 3.3). Logically, an annotation consists of two files (see Figure 3.6). One file, *the data file*, contains the actual data in the annotation. This can be any type of media: ASCII text file, a sound file, a postscript file, images stored in a particular format, or several other formats. These files are created using existing media editing tools for operations such as recording a sound, drawing pictures, grabbing a portion of the screen, text editing, and so forth. KW associates a *link file* with each data file. The link file contains descriptions of the annotations that are made with respect to the data file. For example, if the data file is an image, then the link file consists of records describing all the annotations that were made relative to that file. The records in the link file are described as a set of fields and field-values that include information such as: where the annotations occur, constraints on how the annotation is displayed, time of creation, the author, textual keywords, and several others. The data file is considered a node and the link file specifies links relative to that node and to other annotations (this is only a logical description -- the link files and the corresponding records they contain are not stored in a distributed set of files, but in a small number of large files which are organized for effective database operations).

This simple representation supports the hierarchical grouping of annotations. A link file can contain several annotations, which it serves to group together in the context of the file being annotated. These annotations can involve files, which in turn contain other annotations, providing recursive, hierarchical structures for organizing annotations. This is useful for describing complex relations among several annotations. It is also useful for making templates to direct responses from people. For example, a text data file can have annotations that correspond to where people are requested to input some type of data.

The annotation space consists of all the annotations and files that have been developed. The entire space of annotations can be very large so that it can be partitioned into different subspaces organized in terms of files and directories. Users can select a given directory of annotations or particular annotation files to interact with. This is done via database queries and local link following.

An example query could be to select all annotations authored by Ian, produced after a certain date, which contain the word "convolution." Because KW is based on top of existing facilities, such

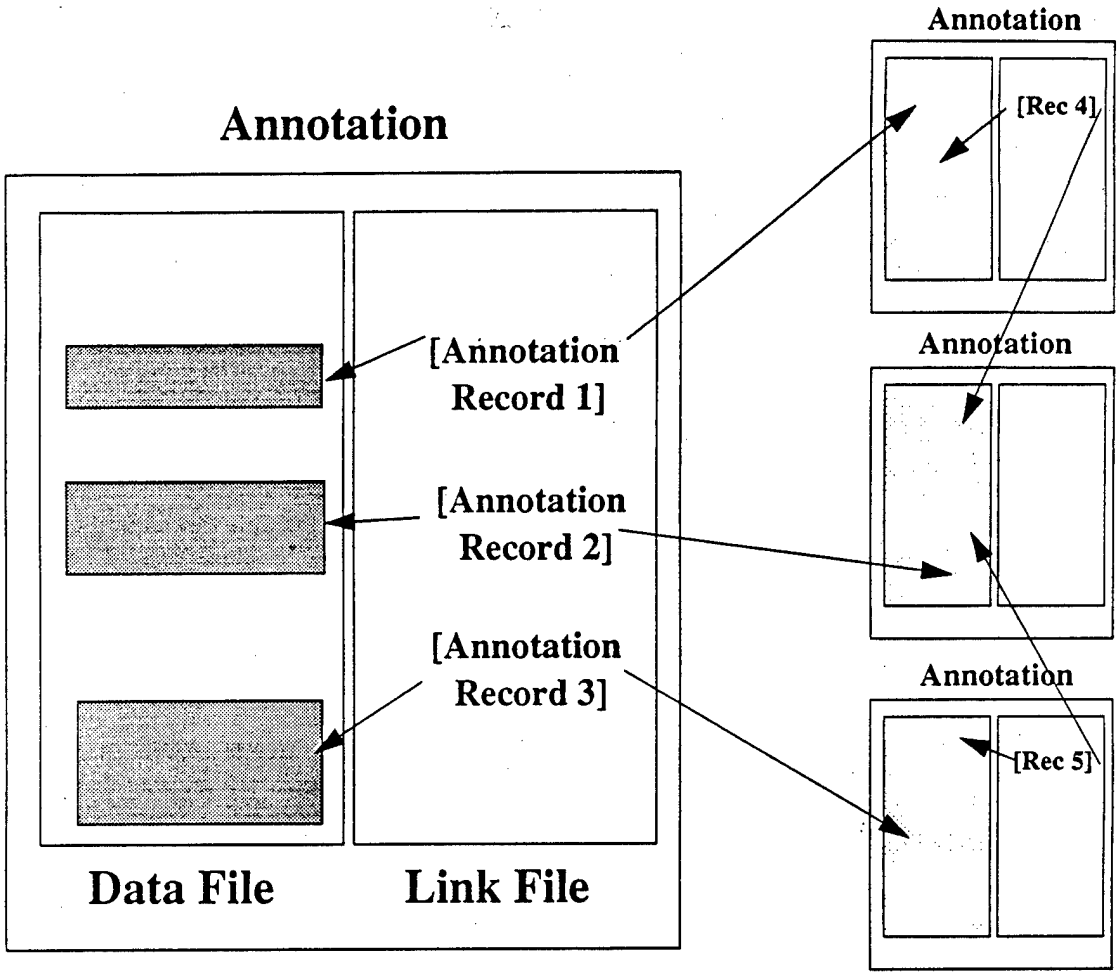

Figure 3.6. Annotation Logical Structure.

queries can involve grep-like pattern matching. Queries are processed by the "Annotation Manager." Associated with a user is a mode description, which is a set of queries that he wants to always be performed either at start-up or at all times. The use of queries to select a relevant set of annotations corresponds to a basic type of navigation. Because queries are done so often, they are supported by an interactive tool, the annotation browser, so they can be performed with limited user activity. The query language is described in section 3.4.

The display of buttons that indicate the location of annotations in a document is handled by the presentation manager. For example, a document may be very heavily annotated, and if all the corresponding buttons were presented to the user, he would be overloaded. The user can select the portion of a given annotation subspace that he wishes to be displayed in a particular way. For example, in viewing a text file, a user can specify that he wants all the annotations of a certain type, made after a certain date, and containing a particular phrase, to be displayed in flashing red in a particular font.

The annotations that are made on a particular file (such as text in an editor or a displayed image) are displayed as mouse-sensitive regions. The display of annotation buttons occurs as an overlay on top of the particular type of media. Buttons are displayed in two different forms: as text in brackets or as a bitmap which can be displayed in an overlay plane. A user can decide to map a selected set of annotations onto several different visual attributes. This is done so frequently that there is an interactive tool to support the selection of display attributes and to keep track of them.

To display an annotation button, the presentation manager uses the record for a given annotation in the context supplied by the annotation-link file. For each of the different types of media that can be annotated, there is an "annotation renderer" that will display the annotation buttons correctly for that type of media. A renderer basically interacts only with the annotation record description (other types of interactions are mentioned in section 3.3.3).

In KW, an annotation which is in a set that satisfies a query is said to be "selected." An annotation whose corresponding data file is displayed in a window is said to be "activated." And an annotation whose button is displayed with respect to a window is said to be "indicated." A "view" is defined as the sets of selected, active, and indicated annotations. The current view is formed by a sequence of queries and can be used to narrow a set of annotations to form a small set of selected annotations. A view is stored as a DBM database of records indexed by the IDs of the annotation records. (IDs are generated for every annotation in KW and are guaranteed to be unique.) Views are stored in a file (DBM format). The browser can store and reload these entities.

Authoring in KW can be generally thought of as the process of adding new text or annotations to an annotation space. There are several ways that one can use KW's already discussed authoring tools.

One way to do authoring (and the way many KW documents get created) is by an author having an idea for a hypermedia document and creating it from scratch. Generally, this type of authoring yields a single document with supporting information as annotations, which are frequently multimedia annotations such as figures, images, and extra references. These types of documents are designed to be read within the context of KW.

A second way to do authoring is to take an already existing document and put it in KW so that it may be annotated by a community of users. An example of this type of document might be a strategic plan for a company, a set of course notes for a college course, or a scanned-in text. Documents can benefit from the collection of knowledge that builds up about the content of the document. This type of annotation usually is a discussion of the document itself.

A third type of authoring that can occur with KW is "guided." In this type of annotation, the author constructs the document in such a way that the reader should respond to questions with annotations of some type (generally textual or programs) which are reviewed later by the author.

Some of the components of KW will now be discussed in more detail.

3.3 Annotations

In this section, the properties and issues involved with annotations are described: the structure and use of the annotation record; some of the different types of annotations currently supported in KW, and issues involved with annotating non-textual media.

3.3.1 Annotation Record

KW annotations are described using a common record format. This is associated with annotations in different media to allow uniform access via database queries and to supply a minimal amount of information for the different annotation renderers to successfully display the annotation buttons. The records have several fields which are described below. It should be noted that the first two fields of this record structure imply the link-structure of the KW document. The value of those two fields are filenames; when KW creates a target file for data (such as when a textual annotation is created), it must be careful to avoid potential concurrence hazards that could arise in naming the file. To solve this problem, mutual exclusion is enforced on the namespace between KW processes running in a network.

Annotation Field	*Value In Field*
Source File	Annotated Document
Target File	Annotation Document
Media Type of Target File	Media Type
Button Location	Source file location of annotation button
Time	Date and Time of the creation of the annotation
Annotation Name	User supplied name of the annotation
Author Name	User who created the annotation
Functional Role	Classification of the purpose of the annotation
Source File Support	Area in the Annotated Document
Target File Support	Area in the Annotation Document
Media-Defined	Button display attributes (such as bitmap)
Source-Changed Flag	Whether the source has been altered
Target-Changed Flag	Whether the target has been altered
User-Defined Keywords	Keywords associated with the annotation

The "functional role" is picked from a defined set of choices; some examples are "todo," "supporting argument," "correction," and "generalization" (these can be extended by users). For now, these classifications allow a browsing program to have some simple ideas about the content of an annotation without resorting to expensive natural language understanding techniques. Eventually, it is hoped that they can be used by the presentation manager for intelligent presentation of a sequence of annotations.

The user-defined keyword list allows users to display and modify a property list on each annotation record. This can be used to store "extra" information that may become necessary to the user at run-time. The use of this feature can be combined with lisp hooks in the implementation of KW to provide easy user customization of KW's behavior.

3.3.2 Types of Annotations

A basic dimension for distinguishing annotation types in KW is the type of media. Several different ones are currently supported: images (in different formats), sound files, text, postscript, programs, and others. A few things are required for a media type to be used. One is that there should be existing media editing and creation tools to create documents. The other is that an annotation renderer has to be developed for the particular media. The renderer understands how to interpret the annotation record to overlay an annotation on top of the particular type of media when it is displayed and can deal with effects such as window resizing and movement.

One type of annotation that was found useful is *reactive annotations*. Using these, a user can interactively grab a portion of a document and make a reactive comment like "good," "unsupported," "confusing," and "add example." Comments are chosen from a user-definable palette of choices, and are intended to allow a reader of a document to quickly make comments to the author of the document. Links that would point to these reactive annotations are not normally displayed by KW when a document is displayed; however, these links are stored in a database. In reading a document, a user can indicate an area of the document and do a query to find other people who commented on the document – perhaps people who DIDN'T indicate they found the section confusing. Authors can use reactive annotations to get feedback. They can interactively select a portion of the document and get a count of the number of different types of reactive comments made on the corresponding portion of the document.

A program annotation in KW is an annotation that is really an executable program. When the link to the annotation is traversed, instead of the annotation being displayed (as it would with a textual or multimedia annotation), the program that is the annotation is started, running in a debugger (gdb). The user may then use normal debugging facilities on this program and may annotate the source code of the program itself. When the source code of the program is displayed (which happens frequently during normal execution of the program in the debugger), the display includes previous annotations made by other users. In some sense, one can think of the entire source code of the program annotation as the "textual part" of the program, since any part of the source code may be annotated as if it were a textual annotation.

KW supports some more convenient interfaces to create certain multimedia annotations. It supports a voice annotation capability so that users who prefer voice input over keyboard can easily do so. Such an annotation is created with a special command in KW, but is displayed in the same manner as any other multimedia annotation. Also, KW supports a "link to screen" capability, which allows the creation of image annotations that are screen dumps. The user invokes a special command in KW to create such an annotation, uses the mouse to select a region of the screen, and an annotation is created that is an image of the region of the screen the user indicated. At display-time, the annotation is displayed the same way as any other image (multimedia) annotation.

3.3.3 Multimedia Annotations

Annotating non-textual media presents a fundamental problem which can be understood by contrast with annotations of ASCII text files. Annotations made with respect to ASCII text files have access to significant semantic information based upon the surrounding text in the source document. For example, words in the area surrounding the annotation can be used in queries and textual searches. Compare this to an annotation of imagery or sound: the semantics is essentially inaccessible. How does the button know that it is laid on top of a picture of a person's face or the corresponding portion of a displayed sound file for the word "peachy"?

To reflect this, KW distinguishes between two different types of annotation: surface annotation and deep annotation. Surface annotations are essentially done by the annotation renderers. They use information in the annotation record to display the annotation as an overlay on top of the

display of the document being annotated. It is sensitive to information contained in the annotation record regarding the position of the annotation, a minimum bounding rectangle as its area of support, and can deal with operations such as when the corresponding window is re-scaled or moved. But it can't access the underlying semantics of the data being annotated.

In deep annotation, the annotation can refer to the semantic content surrounding it. This is directly supported for ASCII text files given the location and area of support of an annotation. For other types of textual displays, such as displayed equations or displays in Rich Text Format, it is necessary to map back to the underlying text. For non-textual media, such as imagery and sounds, this content needs to be explicitly associated with the display of the particular type of media. This involves interactively associating a spatially registered interpretation with displays of different media. Generally, tools for this do not exist. An approach to build them is to take automatic recognition systems for speech and images and produce versions that can be interactively controlled.

KW incorporates a basic capability for deep annotation on spatial data such as, imagery. This is called "spatial index" and is a basic representation used in computer vision (which is concerned with automatic image understanding). A fundamental idea in computer vision is that processing associates a spatially tagged symbolic description with images. The spatial index is a map of pointers to the objects which occupy a given pixel in an image. In this way, when a position is selected, access is given to the actual objects used in the interpretation of the image. This is a memory intensive representation, but it gives us access to the types of representations and geometric database operations used in computer vision applications.

3.4 Queries and Global Navigation

The KW query language is based on a C-like syntax, which is then translated into a format suitable for a TCL interpreter [24, 23] and then applied. The KW database browser contains a TCL interpreter internally, and the interpreter is used for the actual evaluation of the queries.

The simplest form of a query is:

```
operand  comparison  Operator  operand
```

The operands in such a query can be constants or functions. The query can be "passed through" the database of annotations (or currently selected set of annotations) by applying the query to every record of the database, and storing those records for which the query is true. Consider creating a query which would result in selecting all annotations made by a user named "Daryl." Such a query would look like:

```
authorName==''Daryl''
```

In this example, the comparison operator is equality ("=="). The left hand operand is "authorName," a function that computes the author's name from an annotation record. The right hand operand is the constant "Daryl." As each record of the database is evaluated, the authorName function returns the author's name associated with that record and it is compared to the given constant.

Boolean combinations of these queries are also permitted using the normal C operators. For example, if your goal was, "I'd like to see all the questions about our strategic plan which were asked after January 1st," a possible query in the query language would be:

```
mediaType==''ascii''&&sourceFile==''strategic-plan''
                        &&date>=''01 01 1993''
```

Functions in the query language can also refer to the underlying graph of nodes and links formed in the annotation space. Such a function is "distanceFrom," which computes the distance from a node (in hops). Given a set of annotations, a query of the following form computes all the annotations reachable in 2 link traversals from the current set: distanceFrom=="2". Finally, some query language functions exist to modify the display of the results of other queries. Sorting functions are examples of this type of query.

Although this query language is relatively straightforward (and quite familiar for C programmers), it was decided early on that a graphical interface to query creation would be necessary for KW to be broadly used. To support this, a palette of commonly used comparison operators (such equality, inequality, less than, etc.), Boolean operators, and sorting operators in the user interface of the KW database browser is provided. Further, all the cells in the KW database browser respond to mouse clicks by inserting their value as a constant into the current query. Also, the names of all the fields in the browser are mouse-sensitive and if clicked on, respond by inserting a function into the current query which does a selection on that field.

3.5 Presentation Manager

The presentation manager has two parts, the database mapper and the document renderers. The database mapper is part of the database browser and controls display attributes of link presentation. The document renderers are small applications (generally written in TCL) that know how to display documents of a particular media type and annotation markers on these documents. Further, these document renderers understand a simple protocol in which they exchange information with the database browser about the documents and their annotations.

KW has commands to direct the presentation manager to display the annotation buttons. These commands are generally not visible (in the textual form) to the user since he interacts with them via the KW database mapper. These commands provide a way to map a selected set of links onto particular display attributes. Presentation commands can control the color, font family, point size, window location, and font modifiers (such as italics, boldface, flashing, etc.).

Most of these attributes are selected via the database mapper once the user has created an interesting set of links to view via database queries. When the user presses the "Apply Mapping" button on the browser, a protocol message is sent to each document renderer, informing it that the display mapping has changed. At this point the document renderers update relevant portions of their display to reflect the new display attributes of links they are currently displaying.

These display attributes permit the user to garner information about the annotation that can be reached via the button displayed without actually traversing the link (or having the link traversed immediately for the user). Further, it provides the user with contextual information about an annotation with minimal effort. Let us consider again our example of the strategic plan that the user annotated: If the author of the document returns to it later and says, "I'd to see all questions about this document made after January 1 1993. I'd like to see the ones after March 1 in red, those by the user in italics, and the rest in blue." To accomplish such a task requires interactions with the database browser and mapper. A database query could be formed interactively to view all annotations made after January 1, and then the database mapper would be used to map these links onto the color blue. A second query could constrain the selected set to those links made by the user and then these would be mapped (with the database mapper) onto italics. Finally a query could be formed to select those annotations that were made after March 1, and these would be mapped onto the red display attribute.

Conflicts in the display attribute mappings are resolved by giving priority to the last presentation mapping applied; conflicts are only possible within colors or fonts as it is easily possible to make text be both "italic" and "red" simultaneously.

Display attributes are also stored as part of a view. A view contains the display attributes that are tied to a selected set of annotations in the database browser. These attributes are not only the information about link display (such as color, font, etc.) but also information about window displays.

Document renderers communicate with the database browser using a simple protocol. This protocol is built on top of the TK tool kit primitive "send," which implements inter-process communication via the X window property mechanism.

The protocol messages from the database browser to the renderers and their meanings are:

Protocol Message	*meaning*
add document:	add a document to the currently available documents of this media type
render document:	display a document with its annotations marked
mapping notify:	the current display mapping has changed

The protocol messages from the renderers to the database browser and their meanings are:

Protocol Message	*meaning*
new annotation:	a new annotation has been created on a document managed by the transmitting renderer.
applicable annotations:	query for the annotations that are currently valid for a particular document and their display attribute values.
link traversed:	this informs the browser that a link managed by the renderer has been traversed.
renderDocument:	this message instructs the browser to attempt (if possible) to render a different document (potentially not on the sending renderer). This is useful for renderers whose "document" may have supporting documents that need to be rendered at the time the original is rendered.

It is the responsibility of the database browser to be aware of the media types involved and to insure that only documents of the proper media type get sent to the appropriate renderer.

3.6 Implementation Issues

3.6.1 Motivation for using existing tools

One of the main goals of KW was to build on already existing software systems to allow for maximum portability and rapid development, and to allow as broad a user base as possible for KW. Many users will simply not use new systems that require large amounts of new support infrastructure to operate properly.

3.6.2 Tcl and Tk

Tcl (Tool Command Language) is a small, embeddable command interpreter, and Tk is an X Windows toolkit built on top of Tcl. Both of the systems are freely available for a wide variety of platforms. Tcl, by default, understands a variety of programming constructs and simple commands, and it is easily extensible by the programmer to perform application specific commands. By extending the Tcl interpreter, applications can "hook together" the application-specific code with a fully functional command interpreter. Tk is a set of such extensions to Tcl which permits X windows-based user-interfaces to be constructed with Tcl codes.

3.6.3 Knowledge Weasel Implementation parts

Knowledge Weasel's (KW) implementation can be broken into two large pieces, the browser and the renderers. The KW browser includes both the database renderer and mapper and is written in about 1500 lines of C++ code and 3500 lines of Tcl. The Tcl code, however, is generated primarily by an interface builder application (XF). The C++ code is used for performance reasons when interfacing the Tk-based user-interface to the KW database, which is built on the standard UNIX database library dbm.

The document renderers are written almost entirely in Tcl and Tk. The applications generally are built on already existing Tk-based document display objects (called "widgets"), such as the text widget for ASCII text and the photo widget for images. These basic widgets are extended so that they can respond to the protocol described in section 3.5. Renderers can also be built around already existing applications. An example of this is a program renderer currently under development that is based on the gnu debugger. This renderer starts up the debugger on a "program document," when given the protocol message "render document," and then uses the message "renderDocument" to display the supporting source files as ASCII text on the ASCII text document renderer.

3.6.4 Interaction with the lock daemon/Concurrency in KW

What is the lock daemon?

The lock daemon runs on a file server, which exports file system to clients in a network. At this point in time, most vendors provide a "lockd" in their NFS implementation that ships with the workstations. When a client application wishes to have exclusive access to a file that actually resides on a remote server, a "lock" must be requested from the lock daemon for that remote server. After receiving the request, the lock daemon for that machine grants requests for locks in first-come-first-served order. When a client finishes with a lock, it informs the lock daemon and the next client is notified that it has been granted the lock. The granted locks are not preemptable, so clients that die while holding locks are a problem (unbounded wait). The use of "lockd" is not required by NFS, or any other operating system file management facility, so all clients wishing correct access to shared files must participate voluntarily.

The lock daemon and KW

Many instances of KW may run concurrently in a network. To insure correct access to the files used (written) by all instances, KW contacts the lock daemon. KW uses the lock daemon to serialize access to the namespace of annotations (e.g. guarantee that all generated names are unique) and to serialize access to the shared databases of annotations. At the present time, KW locks the entire database whenever a database write is needed, and this could be a performance bottleneck if enough instances of KW were competing for the lock simultaneously. The lock

daemon provides an interface to locking parts of a file, and that should be used in the future to allow some degree of concurrency. Figure 3.7 shows the implementation layers of the Knowledge Weasel.

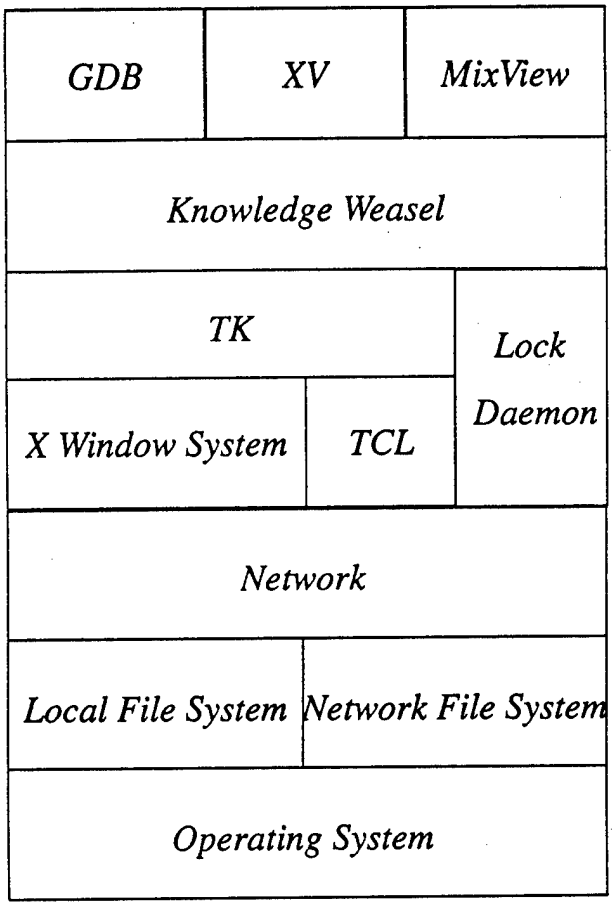

Figure 3.7. Knowledge Weasel Implementation Layers.

3.7 Future Work

- **Comprehension experiments and use of KW in large courses:** An interesting feature of hypermedia is that authoring is still critical. It's clear that books are not going to be replaced any time soon. There is too much value in a coherent linear narrative (in fact it is important to have facilities that will generate these narratives automatically). For example, it is difficult to imagine someone doing a better job of introducing people to quantum electrodynamics than Feynman's QED [7]. A key emphasis for KW is to take advantage of the extraordinary amount of structured knowledge in books and films by using such presentations as a backbone and then to build a society of annotations around it. Studies are being planned on the use of KW with respect to a textbook in a moderate-sized college course. Student comprehension will be compared by using the on-line annotated version of a textbook as opposed to a regular textbook.

- **Deep annotation tools:** Functionally, separating superficial from deep annotation, and providing uniform support for superficial annotation for different types of media, cleaned up the design and development of KW. But many of the interesting issues in hypermedia concern how to effectively annotate non-textual and dynamic media. Currently, a tool for imagery annotation is being developed, based on making the underlying components of computer vision systems interactively controllable by a human so the human can associate an interpretation with images.

- **Incorporation of rule-based and agent-based processing:** There are many places where autonomous processes could help in KW: searching for related annotations, intelligent defaults for the display of annotations, automatically forming views. And as important as annotation is, equally important is anti-annotation for pruning of unused or wrong annotations. An evolutionary incorporation of such capabilities as the processing of the query languages will be extended to autonomous agents. Hopefully a C-based expert system shell (CLIPS) can be incorporated with KW to perform user-directed inferencing over the common annotation records.

Chapter 4

Translational Decomposition of Flow Fields

4.1 Introduction

Chapter 4 introduces a low-level description of image motion called **local translational decomposition (LTD).** This description associates with image features, or small image areas, a three-dimensional unit vector describing the direction of motion of the corresponding environmental feature or small surface area. The local translational decomposition is derived by applying a procedure for processing purely translational motion to small overlapping image areas. This intermediate representation of motion considerably simplifies the inference of motion parameters for ego-motion and can support qualitative inferences for non-rigid motions. First shown is how to compute the LTD from optic flow fields and then how the LTD is used to recover the parameters of rigid body motions.

In previous work [15], a technique was developed to process relative translational motion of a sensor with respect to a stationary environment or independently translating objects. This and related algorithms [4, 11] are based on the strong geometric constraints on image motion in the case of translation – radial motion of image features from a focus of expansion (or contraction) determined by the intersection of the axis of translation with an imaging surface [9, 18]. The technique [15] was based on optimizing a measure that described the quality of feature matches restricted to lie along the radial flow paths associated with a potential axis of translation. The optimization process involved searching over the surface of a unit sphere where each point corresponded directly to a possible direction of translation. The optimization combined the determination of the direction of translation and the corresponding image displacements into a single, mutually constraining computation. It was possible to determine the direction of translation to within a few degrees in small image areas with a few distinctive features.

The translational processing algorithm is extended to work with a general rigid body and other cases of motion by applying the translational procedure to local portions of a flow field. This processing associates a direction of relative environmental motion with the corresponding local portion of a flow field and also with an error measure reflecting the validity of the translational approximation. This description of image motion is called the **local translational decomposition (LTD)**. Computing the LTD begins by decomposing a flow field into small overlapping neighborhoods and then approximating the motion for each neighborhood as being produced by translational motion of the corresponding portion of the environment. This approximation associates a unit vector describing the direction of environmental motion with local portions of a flow field. Each unit vector has an associated fit-value reflecting the validity of the translational approximation.

The LTD is a low-level representation of environmental motion, which considerably simplifies the recovery of the sensor motion parameters. The local directions of motion and the corresponding error measures are used as constraints to determine the actual parameters of motion and to recover

the structure and layout of environmental surfaces. This is broken into four cases. For motion constrained to a plane of a known orientation (see Section 4.2.1), the local translational approximation is recovered directly from the intersection of flow vectors, with the horizon line determined by the plane of motion. For motion constrained to a plane of unknown orientation (see Section 4.2.2), all of the computed LTD vectors must be perpendicular to the normal of the unknown plane. This constraint leads to a direct fitting procedure to recover the plane of motion. For motion relative to locally planar surfaces (see Section 4.2.3), the combination of local planarity and ridigity is used. For arbitrary motion, rigidity between environmental points is used to recover motion parameters from a small number of image locations (see Section 4.2 and 4.3.1).

The remainder of this section introduces the notation used throughout this chapter. Section 2 describes how the local direction of translation is estimated from a flow field and cases of motion for which this is particularly robust. Section 3 describes how the parameters of relative sensor motion can be recovered from the estimated local directions of translation. Section 4 discusses computing the local translational decomposition directly from real image sequences without the initial extraction of optic flow and discusses areas for future work.

4.1.1 Notation

The coordinate system used in this report is shown in figure 4.1. The origin of this right-handed coordinate system lies at the focal point of the camera. The image plane is parallel to the xy-plane and is centered on the point $(0,0,f)$, where f is the focal length of the camera. A three-dimensional environmental point will be referred to as $p_{i,j} = (x_{i,j}, y_{i,j}, z_{i,j})$. The corresponding image point is $\tilde{p}_{i,j} = (\tilde{x}_{i,j}, \tilde{y}_{i,j})$. The first subscript i is used to differentiate between points. The second subscript denotes the time interval. Thus, $p_{i,j}$ refers to the ith point at time j. A three-dimensional displacement which transforms $p_{i,j}$ into $p_{i,j+1}$ forms a vector. This vector will be referred to as $v_{i,j}$. The corresponding optic flow vector on the image plane is $\tilde{v}_{i,j}$. In section 4.2, a method for estimating $v_{i,j}$ is presented. This estimated vector will be referred to as $\hat{v}_{i,j}$. If $\hat{v}_{i,j}$ is correct, it will be parallel to $v_{i,j}$, but its depth will be unknown. $\hat{v}_{i,j}$ can be positioned anywhere along the rays of projection which pass through $\tilde{p}_{i,j}$ and $\tilde{p}_{i,j+1}$. Unless specified otherwise, $\hat{v}_{i,j}$ will be positioned at the image plane.

The motion of the camera can be described by six parameters. Let $r = (r_x, r_y, r_z)$ denote the axis of rotation, and $t = (t_x, t_y, t_z)$ the direction of translation. It is assumed that the axis of rotation passes through the origin of the camera coordinate system. The magnitude of r is equal to the angle of rotation, and t is a unit vector.

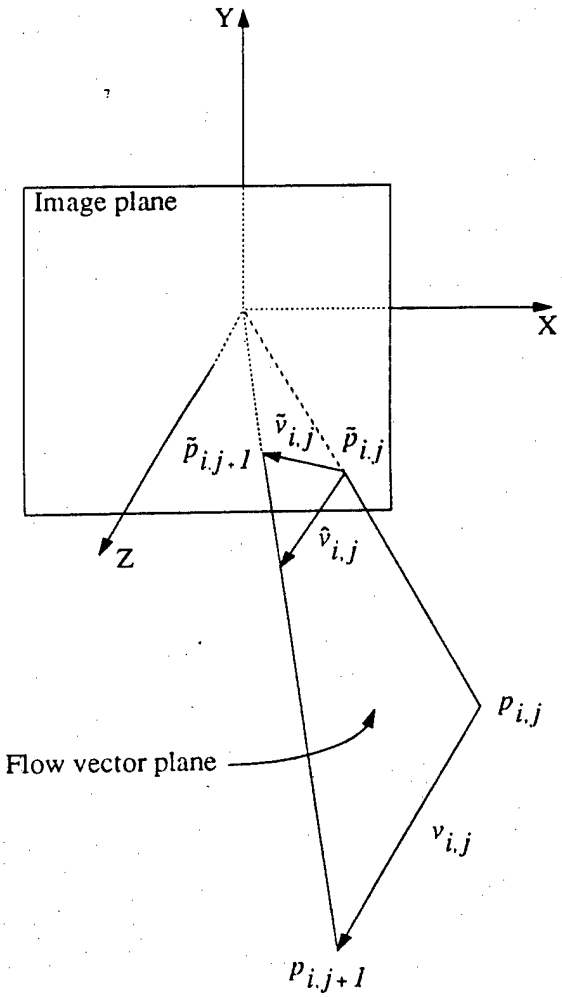

Figure 4.1. Camera coordinate system.

4.2 Estimating Local Translation

This section shows how to determine an axis of translation consistent with a local portion of a computed flow field. Section 4.4 briefly discusses how to compute this directly from textured images without the initial extraction of a flow field.

Figure 4.1 shows that the plane formed by a flow vector and the focal point of the camera must include the estimated local translation vector (this is referred to as the *flow-vector plane* for a given flow vector). In the case of purely translational motion, the estimated local translation vector will be the same for all flow vectors in the neighborhood. Therefore, the estimated local translation vector is the vector that is parallel to all of the flow vector planes in the neighborhood. This observation leads directly to a method of solving for the estimated local translation.

The plane formed by $\tilde{v}_{i,j}$ and the focal point of the camera must include $\hat{v}_{i,j}$. Let this plane be designated by its normal $n_{i,j}$.

$$n_{i,j} = \tilde{p}_{i,j} \times \tilde{p}_{i,j+1} \tag{4.1}$$

Since $n_{i,j}$ is perpendicular to $\hat{v}_{i,j}$

$$n_{i,j} \cdot v_{i,j} = 0 \tag{4.2}$$

In the case of purely translational motion, the direction of $\hat{v}_{i,j}$ is constant for all i. Therefore, Equation 4.2 can be rewritten as

$$n_{i,j} \cdot \hat{v}_j = 0 \tag{4.3}$$

where $\hat{v}_j = \hat{v}_{i,j}$ for all i. This equation is linear with three unknowns, and can be solved using a least squares technique.

An error measure is used to evaluate the validity of the local translation approximation. The error measure used is the average, taken over the local neighborhood, of the angle between each flow vector plane and the local translation. Using the normals $n_{i,j}$ from Equation 4.1, the error measure is defined as

$$\frac{1}{m}\sum_{i=1}^{m}\left|\sin^{-1}\left(\frac{n_{i,j}\cdot\hat{v}_j}{\|n_{i,j}\|\,\|\hat{v}_j\|}\right)\right| \tag{4.4}$$

where m is the number of flow vectors in the local neighborhood. Alternatively (and with greater expense), this measure could be optimized directly by a search procedure to determine an axis of translation.

In general, $\hat{v}_{i,j}$ is not constant for all i. However, in local areas $\hat{v}_{i,j}$ is approximately constant. For example, in Figure 4.2, points that are nearby on a line segment are shown to have approximately the same local translations when the line is rotated about its midpoint. Points near the axis of rotation would not have a good translational approximation as would be reflected in the corresponding error measure. Note that if the motion is composed of both a rotation and translation, the approximation will also be effected by environmental points at different depths, especially at occlusion boundaries. Since the flow vectors in the area of an occlusion boundary will not consistently emanate from a focus of expansion, the error measure given in Equation 4.4 returns a high value in these areas. Using the error measure, the unreliable occlusion areas can be avoided when computing the parameters of motion.

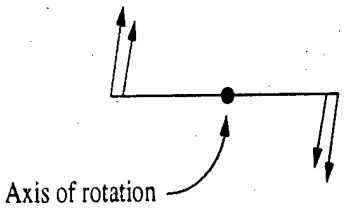

Figure 4.2. Local translation associated with a rotating line.

Figure 4.3 shows the flow field for a scene containing multiple depths and undergoing an arbitrary motion.

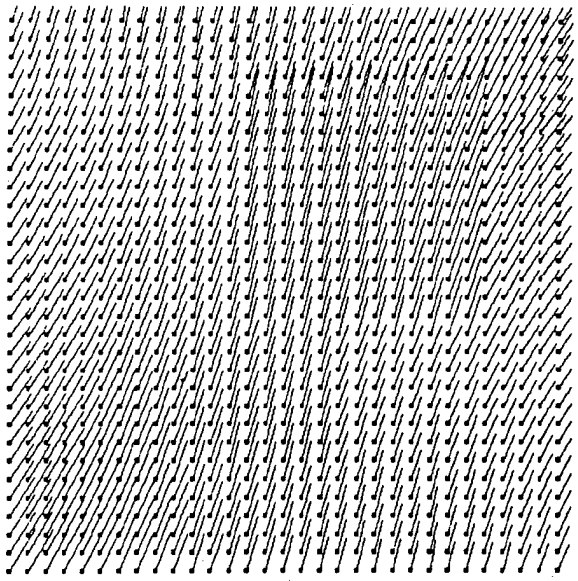

Figure 4.3. Flow field for an image containing occlusion.

The error function derived from this flow field is shown in Figure 4.4. The scene contains two planes that occlude a planar background, as well as each other. The planes, as well as the background, are skewed with respect to the image plane (i.e. the planes are receding in depth). The locations of the occlusion boundaries are obvious from the figure.

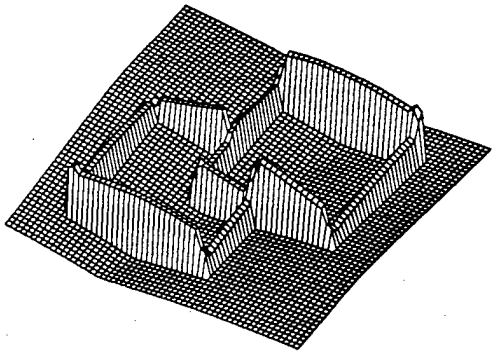

Figure 4.4. Error function for an image-containing occlusion.

The method of LTD estimation discussed above was tested on several synthetic optic flow fields like the one shown in Figure 4.5. This flow field is the result of a rotation of 5.73° about the axis (5, 4, 1), followed by a translation of (100, 25, -75). All units are given in pixels. The field of view of the camera is 90° in both the X and Y directions. The image is 63 x 63, and the focal length is 31. The rectangle overlayed on the flow field represents the neighborhood over which the translational approximation is performed.

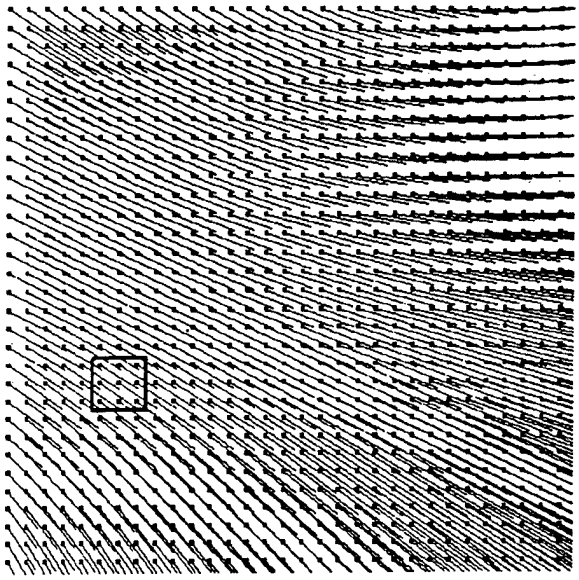

Figure 4.5. Optic flow field for a rotation of 5.73° about the axis (5,4,1) translation of (100,25,-75).

The actual angles between the correct local translational vectors and the approximated local translational vectors at each position in the flow field is shown in Figure 4.6.

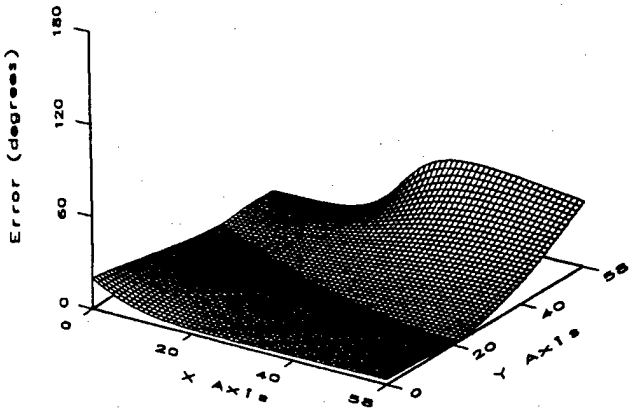

Figure 4.6. Actual errors for flow.

The computed error measure, based upon Equation 4.4, is shown as a surface plot in Figure 4.7. Notice that the computed error measure in Figure 4.7 reflects a strong correspondence between the approximated translational vectors with the least error and the correct translational axes. This correspondence has been found to be typical.

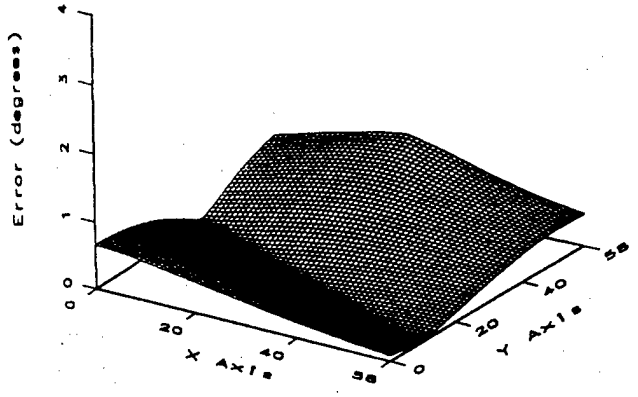

Figure 4.7. Evaluated error measure for flow.

Figure 4.8 (a)-(c) shows the correct local directions of translation with the values of each component displayed as separate intensity plots. Since the translational vectors are represented as three-dimensional unit vectors with each component in the range of -1.0 to 1.0, Figure 4.8 displays the x, y, and z components of the local translation vectors with pure white corresponding to the value of 1.0 and pure black corresponding to -1.0. Figure 4.8 (d)-(f) shows the local translational values that were derived from the optic flow field using the approximation procedure. The derived LTD vector components have been thresholded using the error measure given in Equation 4.4, so that only the best values are shown. These are then used for inferring the overall parameters of motion. The corresponding areas removed by the thresholding are shown by the enclosed white regions which contain a T.

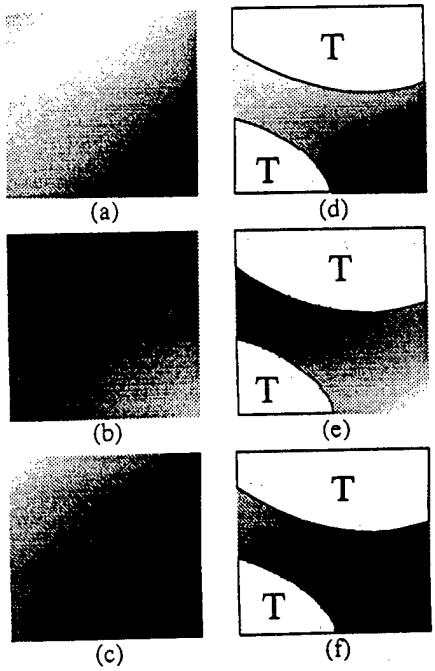

Figure 4.8. LTD vector components of an arbitrary rigid body motion (a) x-component (b) y-component (c) z-component (d) derived x-component (e) derived y-component (f) derived z-component.

4.2.1 Motion Constrained to a Determined Plane

It is particularly simple to recover the local translation from flow fields produced by environmental motion constrained to a determined plane (the normal to the plane is known). In this case, the environmental displacement vector $v_{i,j}$ must be perpendicular to the normal of the plane of motion. We know from section 4.2 that $v_{i,j}$ also lies in the plane determined by its corresponding flow vector $\tilde{v}_{i,j}$ and by the focal point of the camera. The estimated direction of motion lies along the intersection of these planes. The estimated direction of motion $\hat{v}_{i,j}$ can be determined by intersecting these planes. Figure 4.9 shows the geometry, where the plane of motion is positioned so that it intersects the image plane at the base of the flow vector $\tilde{v}_{i,j}$.

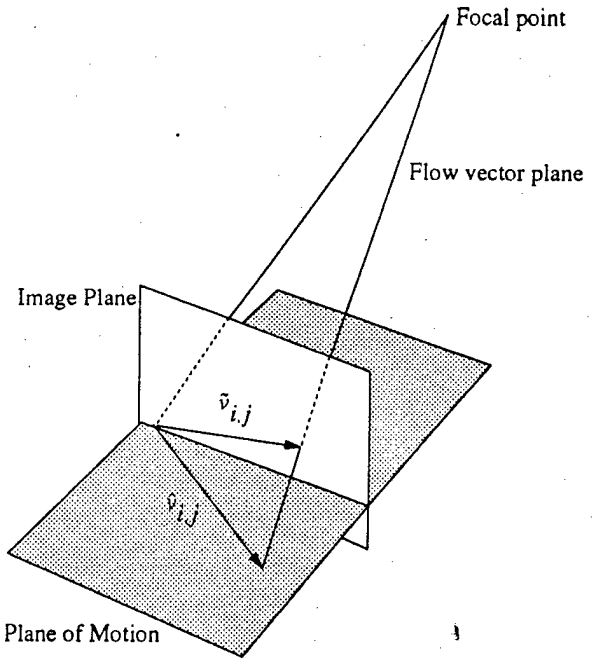

Figure 4.9. Motion constrained to a plane.

In terms of image geometry, this corresponds to intersecting the horizon line, determined by the plane of motion through the focal point, with a flow vector. The point of intersection is a Focus of Expansion for the local axis of translation (or a Focus of Contraction: which depends on the direction of the flow vector relative to the point of intersection). Computing the LTD in this case has been found to give extremely low errors (small fractions of a degree) in the estimated local translations.

Motion constrained to a plane is typical in terrestrial circumstances. Several indoor robotic environments involve robot motion constrained to a plane. In vehicular environments, the translational approximation is usually valid due to limitations in vehicle-turning radii, meaning that the overall motion of a vehicle can be locally approximated as a translation.

4.2.2 Motion Constrained to an Undetermined Plane

Processing in the case of motion constrained to an undetermined plane is similar to that of motion constrained to a determined plane. The only difference is that an estimate of the plane of motion must first be recovered. Using the technique described in Section 4.2, the local translation is computed at each flow vector. Since the motion that produced these local translations is constrained to a plane, each of the local translations must be parallel to this plane. This constraint can be written as

$$\tilde{v}_{i,j} \cdot n = 0 \tag{4.5}$$

where n is a vector normal to the plane of motion. Using this equation, n can be computed by a linear least squares technique.

An example of processing in this case is shown in Figure 4.10 to Figure 4.12. Figure 4.10 shows the flow field produced by a rotation of 4.58° about the axis (-1,1,2), followed by a translation of (120,20,50). Units are given in pixels. This motion is constrained to lie in the plane perpendicular to the normal (-1,1,2). However, the plane is unknown, so initially the local translation vectors must be computed by the method used for cases of arbitrary motion.

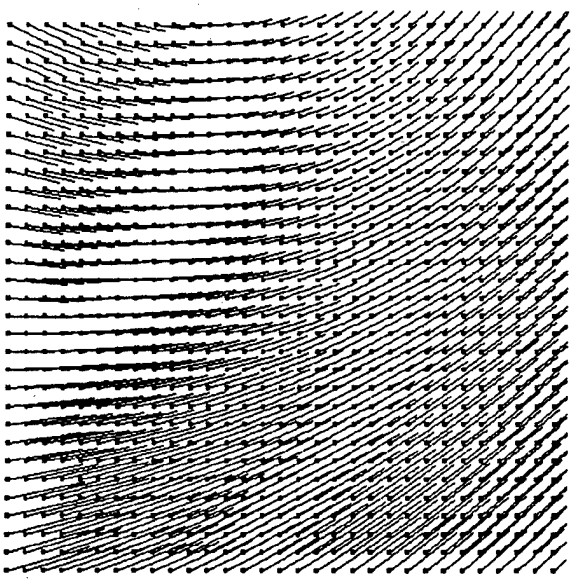

Figure 4.10. Optic flow field for a planar motion.

The angles between the correct local translational values and the derived local translational values shown are plotted in Figure 4.11.

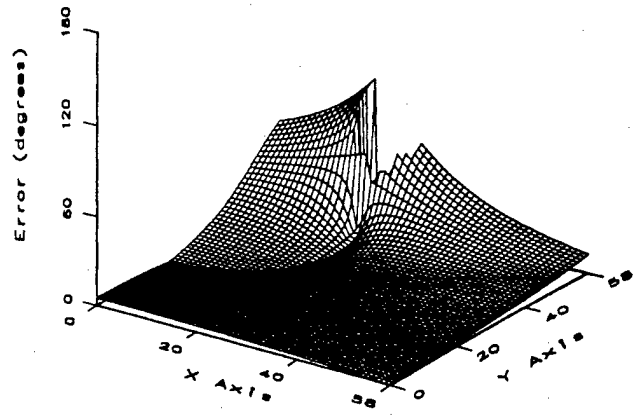

Figure 4.11. Actual errors for an unknown planar motion.

The error measure is shown in Figure 4.12. Since an error measure is associated with each point describing the error of the translational approximation, several positions of minimal error can be selected for use in Equation 4.5. Using the error measure from Equation 4.4, the 15 best local translations were selected for the least squares fit. The recovered plane normal is then (-0.4107,0.4129,0.8129), which is off by an angle of 0.37° from the correct value. This estimate can then be used to evaluate the directions of motion using the technique for motion constrained to a determined plane from the previous section.

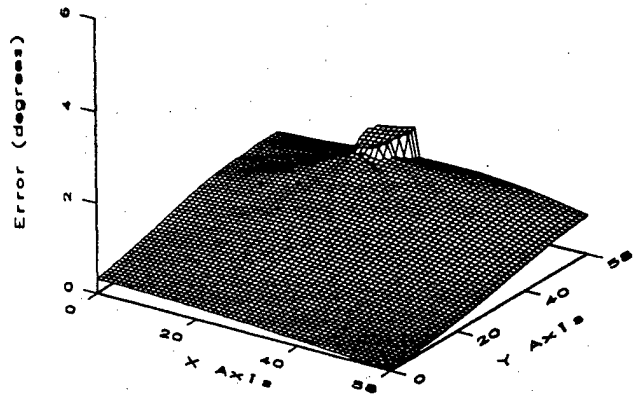

Figure 4.12. Evaluated error measure for unknown planar motion.

The computed directions of motion are then shown in Figure 4.13 (d)-(f). Like the case of motion constrained to a known plane, there is very little error in the derived LTD vectors. The mean angle between derived and actual LTD vectors was 0.176° and the maximum angle was 1.274°.

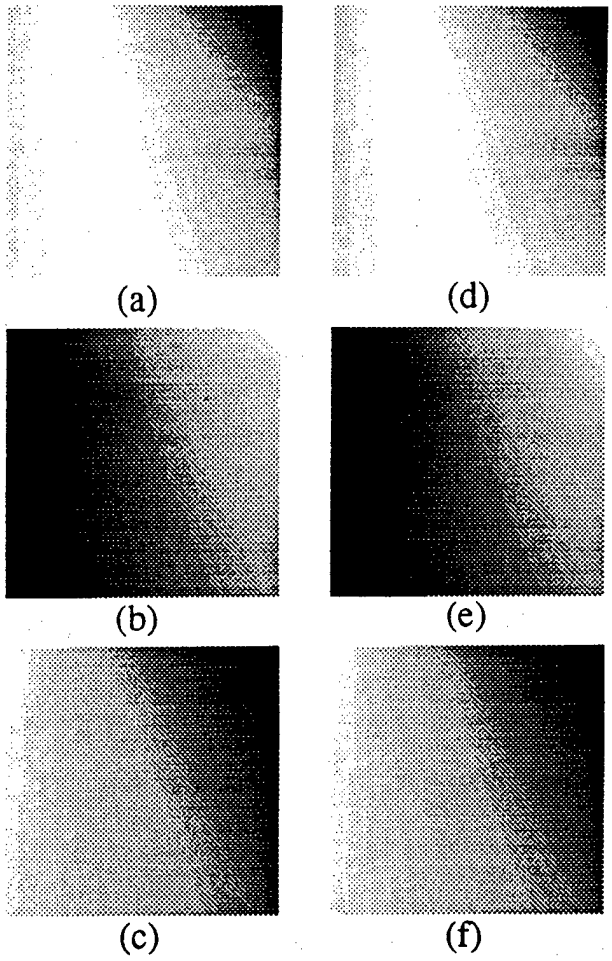

Figure 4.13. LTD vector components of an undetermined planar motion (LTD estimated using the determined planar motion technique) (a) x-component (b) y-component (c) z-component (d) derived x-component (e) derived y-component (f) derived z-component

4.2.3 Local Planarity and Rigidity-based LTD Estimation

Another algorithm for computing the LTD is based on the constraints provided by assuming motion relative to locally planar, rigid environmental surfaces. The algorithm begins by searching over the half-plane defined by a flow vector and the focal point of the camera as shown in Figure 4.1 (this plane is designated a half-plane because only a 180° search is needed). Each candidate LTD vector is used to solve for other LTD vectors in a local neighborhood by making an assumption of surface planarity within the neighborhood. The consistency of this local neighborhood of LTD vectors is then evaluated by calculating the relative depths of the LTD vectors. This results in an error measure that is associated with each candidate LTD vector. The

candidate LTD vector with the lowest associated error is selected as the correct LTD vector. The remainder of this section describes this algorithm in greater detail.

Local Planarity Assumption

Given a candidate LTD vector, the other nearby LTD vectors can be solved. In order to derive a relationship between LTD vectors within a neighborhood, it is assumed that these surfaces are locally planar. In this case, directional derivatives of the LTD vectors along the image plane are constant. Let $\tilde{p}_{i-1,k}$, $\tilde{p}_{i,k}$ and $\tilde{p}_{i+1,k}$ be three collinear points on the image plane. Under the planar surface assumption, the following constraint is:

$$\frac{\hat{v}_{i+1,k} - \hat{v}_{i,k}}{\left\|\tilde{p}_{i+1,k} - \tilde{p}_{i,k}\right\|} = \frac{\hat{v}_{i,k} - \hat{v}_{i-1,k}}{\left\|\tilde{p}_{i,k} - \tilde{p}_{i-1,k}\right\|} \tag{4.6}$$

Letting $\hat{v}_{i,k}$ be the current candidate LTD vector, Equation 4.6 consists of two independent equations and six unknowns. The remaining equations needed to solve for these six unknowns can be provided by the LTD vectors' corresponding optic flow vectors. Figure 4.1 shows that the plane formed by a flow vector and the focal point of the camera must include the LTD vector. This constraint can be written as

$$(\tilde{p}_{i-1,k} + \hat{v}_{i-1,k}) \times \tilde{p}_{i-1,k+1} = 0 \tag{4.7}$$

$$(\tilde{p}_{i+1,k} + \hat{v}_{i+1,k}) \times \tilde{p}_{i+1,k+1} = 0 \tag{4.8}$$

This provides four additional independent equations. Therefore, using the system defined by Equations 4.6, 4.7 and 4.8, one can solve for the neighborhood LTD vectors $\hat{v}_{i-1,k}$ and $\hat{v}_{i+1,k}$.

Error Measure

The final step in evaluating a candidate LTD vector is to construct an error measure from the neighborhood of derived LTD vectors. The relative depth of all the LTD vectors in a 3x3 neighborhood is calculated by positioning the candidate vector at the image plane. Using the depth values, a plane is fit to the neighborhood points. The error measure is defined as

$$\frac{1}{m}\sum_{i=1}^{m}\left\| \alpha_i \tilde{p}_{i,k} - q_{i,k}\right\| \tag{4.9}$$

where α_i is the depth scale factor and $q_{i,k}$ is the point of intersection of the fitted plane and the ray of projection defined by $\tilde{p}_{i,k}$. Section 4.3.1 shows how to solve for the depth scale factor α_i.

An example of processing an arbitrary motion using the rigidity-based method is shown in Figure 4.5 and Figure 4.14. Figure 4.5 shows the flow field produced by a rotation of 5.73° about the axis (5,4,1), followed by a translation of (100,25,-75). Units are given in pixels. Figure 4.14 (a)-(c) shows the correct local translational values as intensity plots of the vector components. Figure 4.14(d)-(f) shows the local translational values that were derived from the optic flow field. Like the case of motion constrained to a known plane, there is very little error in the derived LTD vectors. The mean angle between derived and actual LTD vectors was 0.425°, and the maximum angle was 2.647°.

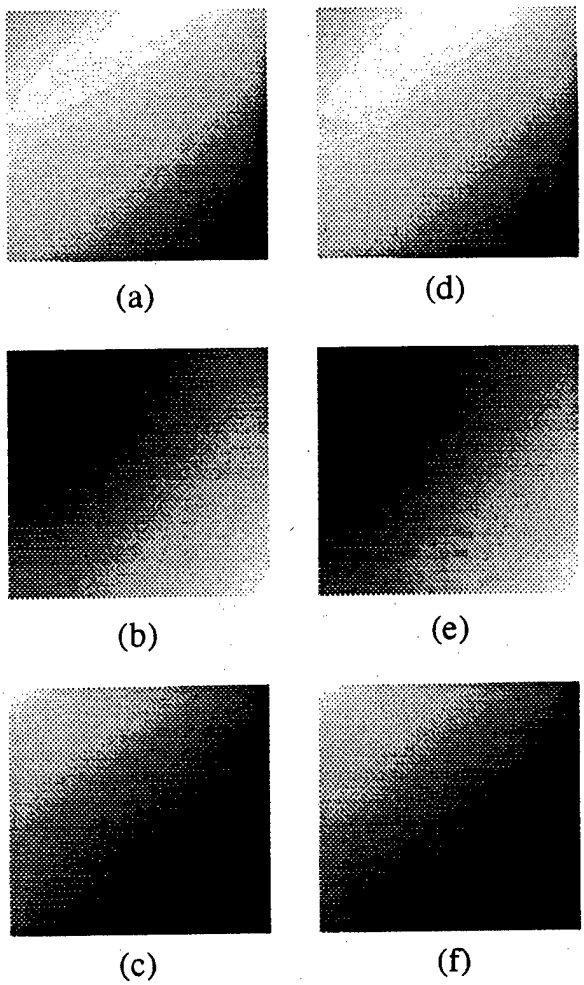

Figure 4.14. LTD vector components of an arbitrary rigid body motion (LTD vectors were derived using the local planar method) (a) x-component (b) y-component (c) z-component (d) derived x-component (e) derived y-component (f) derived z-component

4.3 Inferring Parameters of Motion from the LTD

In this section, a technique is developed to recover the parameters of motion given a flow field and the LTD. The method presented in this section is based upon using rigidity to solve for the relative

depth of environmental points associated with LTD vectors. The key result is that it is possible to infer the parameters of motion using only three determined LTD vectors computed from locations anywhere within the flow field. Thus, the inferencing can be done with a sparse LTD field which may have been strongly filtered by the validity of the measures reflecting the translational fit. Once the relative depth has been determined, the solution for the parameters of motion becomes straightforward.

4.3.1 General Rigidity Constraint

In order to find the parameters of motion, one must first solve the relative depth of the LTD vectors using the rigidity. Once the relative depth has been determined, the solution for the parameters of motion becomes trivial.

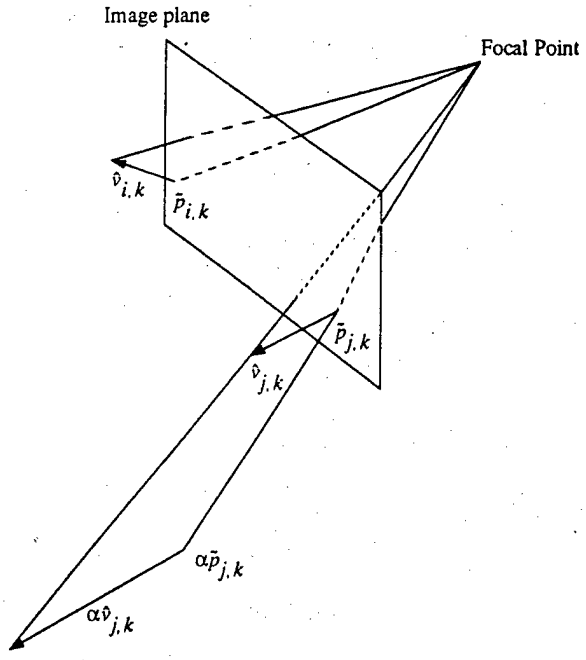

Figure 4.15. Relative depth of two LTD vectors.

Two LTD vectors $\hat{v}_{i,k}$ and $\hat{v}_{j,k}$ are assumed to have undergone identical rigid body motions. One wishes to find the relative depth of these two vectors. Figure 4.15 shows the relationship between the two vectors. One of the vectors, $\hat{v}_{i,k}$, is fixed in depth so that it emanates from the image plane at the point $\tilde{p}_{i,k}$. The unknown depth of the other vector can be expressed as $\alpha\tilde{p}_{j,k}$ where α is some unknown scale factor. Since both of the LTD vectors are the result of the same rigid body motion, it results in the following constraint

$$\left\| \alpha\tilde{p}_{j,k} - \tilde{p}_{i,k} \right\| = \left\| \alpha(\tilde{p}_{j,k} + \hat{v}_{j,k}) - (\tilde{p}_{i,k} + \hat{v}_{i,k}) \right\| \qquad (4.10)$$

Squaring both sides and solving for α, Equation 4.10 can be reduced to

$$\begin{aligned}&(2\tilde{p}_{j,k}\cdot\hat{v}_{j,k}+\hat{v}_{j,k}\cdot\hat{v}_{j,k})\alpha^2-\\&\quad 2(\tilde{p}_{j,k}\cdot\hat{v}_{i,k}+\tilde{p}_{i,k}\cdot\hat{v}_{j,k}+\hat{v}_{i,k}\cdot\hat{v}_{j,k})\alpha+\\&\quad (2\tilde{p}_{i,k}\cdot\hat{v}_{i,k}+\hat{v}_{i,k}\cdot\hat{v}_{i,k})=0\end{aligned} \tag{4.11}$$

This equation is quadratic in α and results in two feasible solutions for the relative depth between two LTD vectors.

4.3.2 Inferring the Parameters of Motion

Once the relative depth between LTD vectors has been determined, the estimation of the parameters of motion is trivial. The problem is equivalent to that of estimating the motion parameters from actual three-dimensional environmental surface positions. A rigid body motion can be expressed as

$$\alpha_{i,j}\hat{v}_{i,j}=r\times\alpha_{i,j}\tilde{p}_{i,j}+t \tag{4.12}$$

where r is the axis of rotation and t is the direction of translation. This expression is linear and can be solved using a least squares technique. The expression consists of six parameters and two independent equations. Therefore, it can be solved using a minimum of three (non-collinear) LTD vectors.

4.3.3 Motion Parameter Inference Results

The rigidity constraints were used to compute the parameters of motion from the derived LTDs presented in Section 4.2. The results are shown for the case of arbitrary motion, motion constrained to a determined plane, motion constrained to an undetermined plane, and the rigidity-based method applied to arbitrary motion. In the previous section, it was noted that the parameters of motion can actually be estimated using only three LTD vectors. The feasibility of estimating the parameters of motion from a minimal set of data is demonstrated in the results presented below.

Motion Constrained to a Determined Plane

In the case of motion constrained to a determined plane, the LTD vector estimates tend to be highly accurate over an entire flow field. Typically, when using three LTD vectors selected at random from the derived local translations, the estimate of the axis of rotation and translation almost always are within a degree of the correct axes, and the angle of rotation is determined to within a hundredth of a degree.

Motion Constrained to an Undetermined Plane

The case of motion constrained to an undetermined plane is similar to the case of motion constrained to a determined plane in that the LTD vector estimates are very good over the entire image. Three LTD vectors were selected at random from the derived local translations shown in Figure 4.13. The estimate of the axis of rotation was off by 0.99°, the angle of rotation was off by 0.04°, and the direction of translation was off by 0.83°.

Local Planar Method

The rigidity-based method presented in section 4.3.1 is also capable of accurate LTD estimates over the entire flow field. Three LTD vectors were selected at random from the derived local translations shown in Figure 4.14. The estimate of the axis of rotation was off by 2.26°, the angle of rotation was off by 0.18°, and the direction of translation was off by 2.84°.

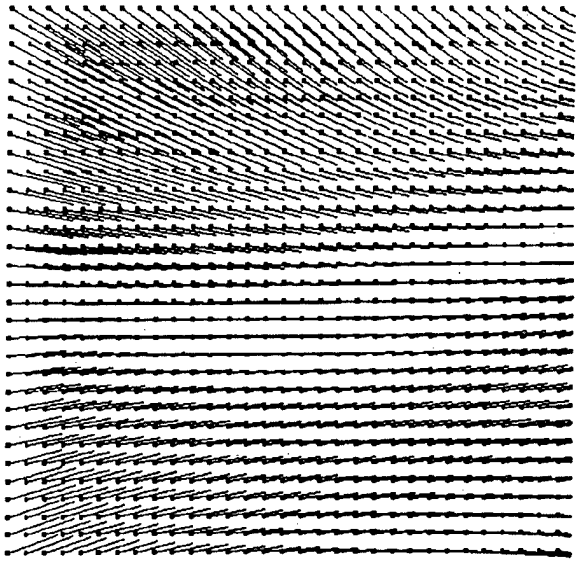

Figure 4.16. Optic flow field for motion relative to a curved surface.

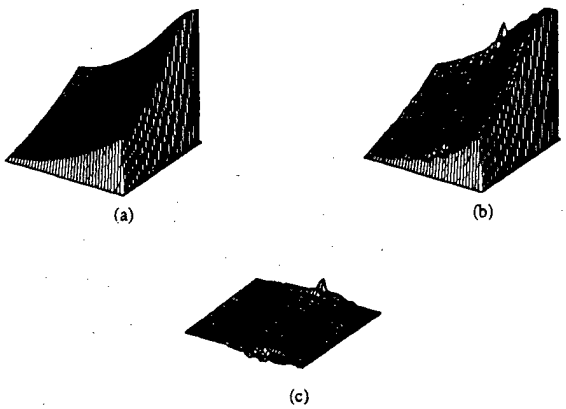

Figure 4.17. (a) Curved surface (b) Reconstructed surface (c) Error.

The camera was moved about a randomly curved surface. The optic flow field produced by this surface is shown in Figure 4.16. The three-dimensional environmental surface was reconstructed from this flow field. Figure 4.17 (a) shows a plot of the original surface. Figure 4.17 (b) shows the results of the surface reconstruction and Figure 4.17 (c) shows the resulting error in the reconstruction. The surface shown in this example is not planar. However, the reconstruction is fairly accurate, despite the violation of the planarity assumption. Experiments indicated that surfaces which are approximately planar in a local neighborhood can be successfully reconstructed.

Therefore, any continuous surface can be reconstructed, given an appropriate density of optic flow vectors.

Arbitrary Motion

Using the error measure shown in Figure 4.7 and the derived LTD vectors shown in Figure 4.8, the three best LTD vectors were selected and used to compute the parameters of motion. The estimate of the axis of rotation was off by 8.13°, the angle of rotation was off by 1.09°, and the direction of translation was off by 12.02°. In the previous section, it was shown that the minimum number of LTD vectors that can be used to estimate the parameters of motion is three. However, by using a larger set of LTD vectors in a least squares procedure the results were more accurate. For example, when the ten best LTD vectors were used, the axis of rotation was off by 3.65°, the angle of rotation was off by 0.44°, and the direction of translation was off by 9.32°.

4.4 Future Work

LTD has been introduced as a low-level representation of environmental motion, which can simplify the inference of motion parameters from optic flow fields. It has been found that this is particularly robust and simple for cases of motion constrained to a determined or undetermined plane, and motion relative to locally planar surfaces. In addition, it is possible to infer motion parameters from sparse LTDs.

Areas for further work include:

- Develop criteria to determine the best set of estimated local translation vectors to estimate motion parameters in order to take advantage of the limited number of points for which the local translation needs to be determined to infer motion parameters.
- Investigate local translational analysis by using multiple cameras and longer image sequences.
- Use LTD, which is similar to an array of localized looming detectors, to determine whether things are coming towards or away from an observer at a particular image position. It may be possible to use such a distributed representation of motion relative to environmental surfaces to control navigation and other behaviors directly, without the inference of motion parameters from the LTD.
- Use local translation approximation as a criteria for computing flow to determine the LTD directly without the initial computation of a flow field. In the experiments presented above, a uniformly dense flow field of high resolution was assumed. The translation procedure developed in [15] was not applied to computed flow fields, but to successive images for which interesting points had been extracted from the initial image. Given distinctive features (at least two), it was possible to compute the direction of translation in a small image area. This use of the translational procedure can be seen as a local constraint to determine image displacements, such that the corresponding environmental motion can be interpreted as being translational. For ego-motion, this wouldn't require computation over the entire flow field since only three LTD vectors are needed. Where the translational approximation is poor, there will be a large value in the error measure reflecting weaker confidence in the validity of the approximation.

Chapter 5

An Interactive Model-Based Vision System for Vehicle Tracking

5.1 Introduction

Chapter 5 describes a general architecture for an interactive model-based vision system and its application for vehicle tracking. A human specifies a limited amount of information, which establishes a context for autonomous interpretation of images obtained by a telerobot. Object models are described by constraints specifying necessary geometrical properties and relationships between objects. The use of constraints allows for flexible object instantiation. A user can indicate a vehicle , and this directs perceptual processing routines to determine the corresponding local surface orientation and roads, or the user can instantiate a road segment to direct the extraction and tracking of vehicles.

Efforts to develop intelligent and autonomous systems for operation in complex, natural domains have been largely unsuccessful to date, in spite of continued advances in the underlying technologies. There remain unresolved and fundamental difficulties in terms of the necessary computational power, the required complexity of perceptual systems that can operate in outdoor environments, and the corresponding complexity of planning and reasoning systems. A recent framework addresses many of these problems by stressing the importance of telerobotic and interactive systems [25, 26]. This is a realistic approach to fielding advanced technology in the short term. Also, it provides a long-term framework for developing autonomous systems. An interactive, semi-autonomous system can significantly amplify the capabilities of a human, and can yield an evolutionary approach as autonomous system capabilities are developed and begin to replace human-controlled functions.

The approach described here is to develop a model-based vision system that a human can interactively control. The human uses this to rapidly interpret sensory information from a potentially distributed team of telerobots. The resulting interpretation is a model of the world that the telerobots can refine, use to control their behavior, or report back to a human. In this way, the human directs the telerobots by initializing and constraining their processing. Communication between the robot and the human can then take place in the context of a shared model of the world, which makes possible infrequent, semantically meaningful, and low bandwidth, communication.

The particular system presented is for tracking vehicles in outdoor scenes. A human can manipulate models of objects, such as terrain surface patches, roads, and different type of vehicles, to interpret imagery from a telerobot. Once an interpretation is in place, the telerobot can autonomously refine and extend the interpretations, detect and track vehicles, and report back to a human about unusual occurrences or behavior that cannot be accounted for. For example, a human will indicate that a particular area is a road. The vision system will then track movement along the road and fit a constraint-based description of a vehicle to this movement. The system can determine that a vehicle has just gone off the road (or that it is behaving inconsistently with respect to the model of a vehicle).

We begin by reviewing the basic architecture of the interactive model-based vision system, and then detail some of its critical components involving object models and perceptual processing.

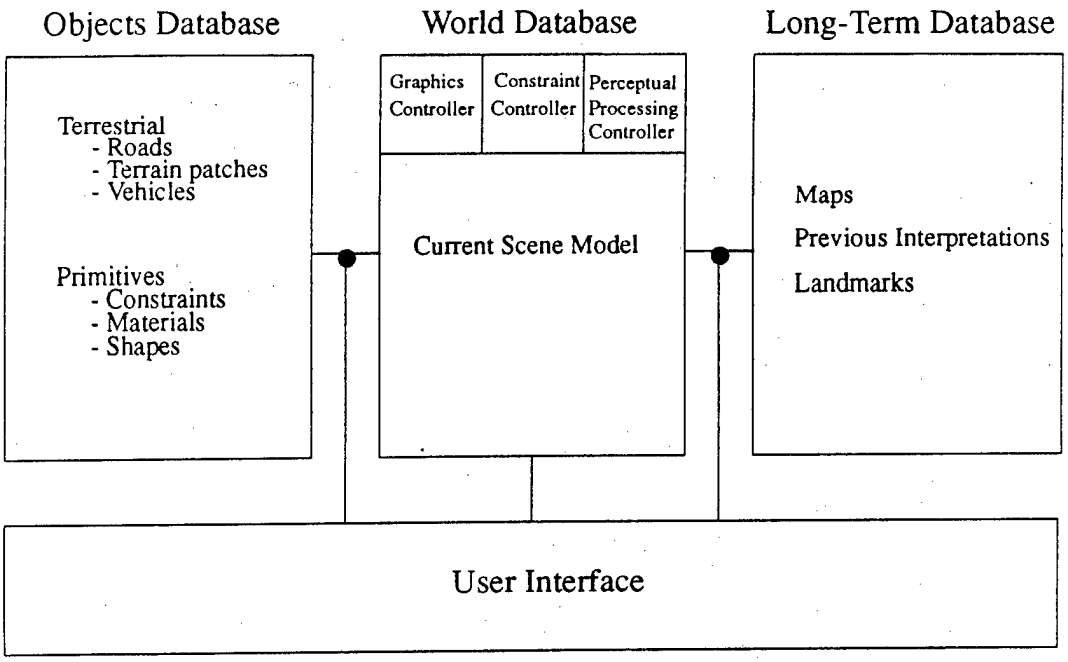

Figure 5.1. System architecture.

5.2 System Architecture

The underlying architecture is shown in Figure 5.1. It is built around three major databases that a human can access and manipulate through a user interface. The basic task of the human is to access models of the various types of objects stored in the **Object Model DataBase,** along with information describing maps, landmarks, and previous interpretations in the **Long-Term DataBase,** to build an interpretation of the current scene which is stored in the **World Model DataBase**. For example, the human is presented with images from cameras on the telerobot. He can use priori maps to align landmarks and terrain features from these maps with the images. He can also access the three-dimensional and physically based models of objects and position them with respect to the world model. As he does this, the models are projected back against the images obtained from the telerobots for interactive control and to initiate processing.

The **Object Model DataBase** contains generic models of objects, relationships, and events for terrestrial scenes. This involves objects such as terrain patches, roads, vehicles, and gravity. Two different types of objects are distinguished: **Primitives,** which correspond to basic entities and relationships used to describe and represent, and **Terrestrial Objects,** which correspond to the conventional objects found in the world such as roads and cars. Primitive Objects describe characteristics such as shape constraints, material composition, and relationships between parts. The representation of objects for an interactive vision system is more complex, though related in many ways, to those used in CAD/CAM and geometric modeling packages, because they will be

manipulated for autonomous processing and reasoning. Thus, in addition to describing its shape, the model of a car needs to include that a car is acted on by gravity and will have a preferred type of orientation and attachment with respect to the ground surface. Object models are described by sets of constraints [2, 14, 19, 22] that must be satisfied. A simple constraint is that the value of some parameter associated with an object model is bounded. More complicated constraints deal with relations between objects. The human will, in general, specify a limited amount of information for an object, and the system will use the constraints and associated processing actions to refine the instantiation of an object.

The **World Model DataBase** describes the three-dimensional world of objects and situations surrounding the telerobots. It is initially formed by the human-accessing models in the object database and instantiating them. There are three types of controllers associated with the World Model DataBase. The **Constraint Controller** checks for consistency in the world model. The constraint controller uses the constraints that define an object or relationship to refine an instantiation or to find a violation or inconsistency and ask the human for help. The **Perceptual Processing Controller** extracts information from images and sensors on the telerobot. The constraints in an object model specify the types of processing that are necessary to obtain this information. When the human indicates that a road is located somewhere, this constrains the type of tracking and feature extraction processes that are used. The corresponding image areas are isolated, and the type of segmentation or tracking procedure corresponding to the material class and distance of the object is applied. The **Graphics Controller** deals with interactive scene measurements and the presentation of the world model to the user. Thus, when he accesses a model of a vehicle, he is presented with a cartoonish 3D vehicle template which is back-projected onto the image being interpreted.

The user interface is currently based upon windows for displaying imagery and graphical overlays and upon text-based browsers for inspecting entities in the database in detail. This basic level of interface can be quite tedious to work with and its future role will be to serve as a debugging tool. An intermediate, near-term system interface will use a more natural set of tools, such as 3D hand/finger position sensors and voice input. Using these, the human will actually have a sense of reaching into the database of models, grabbing something, and then placing it into the world model. In the eventual system, the world model and the sensor input from the different telerobots could be presented to the human as a virtual reality in which the human can be embedded in the world model itself.

5.3 Object Models

Models have been developed for objects corresponding to gravity, the immediate ground plane surrounding the camera from which an image is obtained, terrain patches, and a generic vehicle along with constraints describing relations for attachment, alignment, and coincidence. There is a constraint propagation mechanism to determine consistency of relationships between these types of objects in the world model database.

Different types of road models for two and three dimensions are used. The two-dimensional road model is a sequence of connected parallel line segments for the road boundaries and/or the center-line of the road. This is used to indicate and mask images areas that are adjacent to the road. The three-dimensional road model is a connected sequence of segments with 3D coordinates and associated road width information with constraints on allowable orientations with respect to gravity and adjacent terrain patches. Different material properties can be associated with the roads, but this currently isn't used by the segmentation and feature extraction procedure.

The generic vehicle model is an oriented box with an indication of where the track/wheel area of the vehicle is, where the engine is positioned, and where the cab area is. The scale and relative position of these is parameterized and can be specialized for different types of vehicles. There are scale and orientation constraints on all of these components, as well as for relative position-to-ground surfaces and gravity (see Figure 5.2).

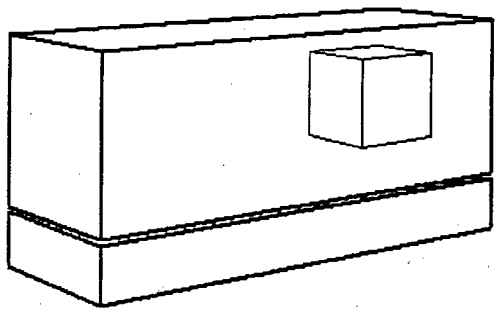

Figure 5.2. Perspective view of the 3D vehicle model.

5.4 Perceptual Processing

Image processing and tracking procedures are organized in terms of the type of information they depend on and can extract. One type of tracker depends on a two- or three-dimensional road model and can yield information to instantiate a vehicle model. An instantiated vehicle model constrains the extraction of features. These features satisfy the requirements of another type of tracker that can determine a scaled three-dimensional trajectory for extracted image points. The information determined by this tracker can in turn be used to determine a three-dimensional road model, and also to refine the attributes of an instantiated vehicle model. As a result, the flow of information and processing varies based upon the state of the current interpretation. The current processing routine consists of three types of trackers, along with restricted segmentations and interest operators, that are applied when a vehicle model is instantiated.

5.4.1 Difference Tracker

The difference tracker operates with respect to an instantiated two- or three-dimensional road model. It determines regions above the indicated road areas that are changing over time and also are moving in a consistent direction (not necessarily along the road). It determines information to instantiate a vehicle model by finding the front and back (or only the back or the front) of a vehicle. If a three-dimensional road model has been instantiated, it can further constrain the dimensions of the generic vehicle model instantiation. It also restricts the extraction of features for the local translational tracker (Section 5.4.2), which can in turn recover the direction of motion of the vehicle, whether it is turning, and the corresponding direction of motion relative to the road.

The first step in the difference tracker is to reduce the image noise by convolving consecutive images in a motion sequence with a low-pass filter. The images shown in this chapter were smoothed using a Gaussian filter. If no models are present, the entire image must be convolved with this filter. However, given a 2D road model, the filter is only convolved with pixels that are above the road. The road model shown in Figure 5.3 is used to constrain the smoothing process.

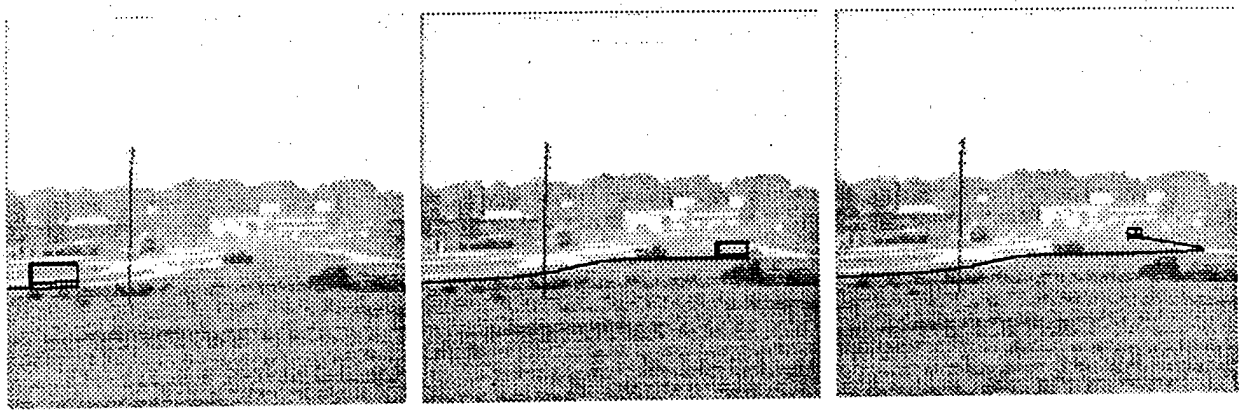

Figure 5.3. Interactively driving the vehicle to form the road model.

Once the images have been smoothed, the algorithm begins to search for areas of motion that lie near the road. This is accomplished through image subtraction. Pixels from temporally consecutive images that are situated near the road model are subtracted. If the result of this subtraction is greater than a threshold, the environmental object corresponding to this pixel position is assumed to have undergone motion. This pixel is marked as a motion pixel, and a region growing process begins.

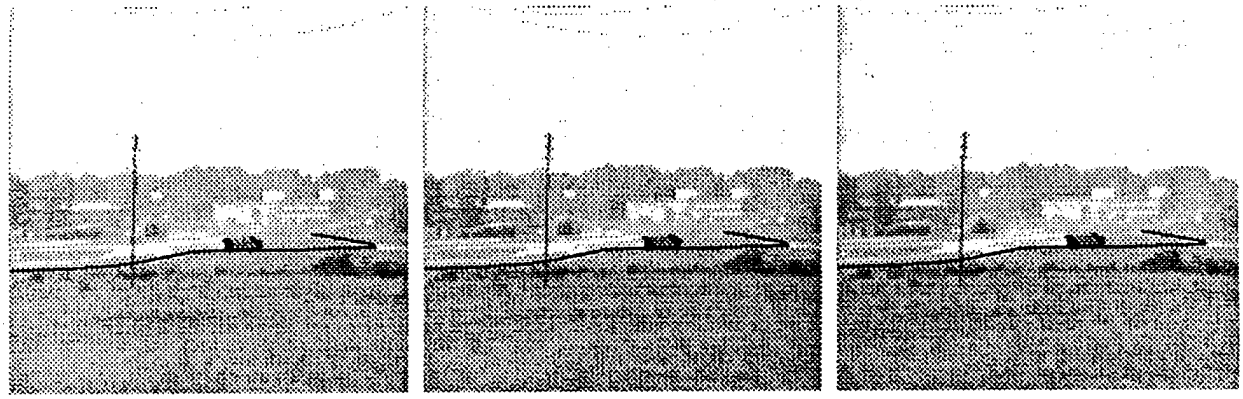

Figure 5.4. Areas of motion and vehicle position found through differencing.

An object traveling along the road may extend some distance from the road (i.e. the object could be very close to the camera, in which case it would appear to be quite large). The search for all areas of motion associated with an object is accomplished through region growing. Once a pixel near the road has been identified as a motion pixel, its neighbors are also examined by using the subtraction technique discussed above. If any of the neighboring pixels contain motion, their neighbors are also examined. This recursive procedure continues until no more motion pixels can be found. An

example of this extraction of the areas of motion is shown in Figure 5.4. Once the areas containing motion have been identified, the centroid of these areas is located. Over time a two-dimensional trajectory can be constructed.

5.4.2 Local Translation Tracker

Moving vehicles can often be treated as rigid objects that are translating over short periods of time. For example, as a vehicle goes around a curve, because of turning radii constraints, the axis of rotation is often far away from the vehicle itself, and the vehicle motion can be treated as a sequence of small translations corresponding to tangents of the curve of motion. The local translation-based tracker determines the direction of motion of a set of extracted image points over time, and fits their motion to an estimate of the current direction of motion of the corresponding vehicle in three dimensions. Essentially, it determines the direction of motion of a set of environmental points over time. The effect of this tracker can be visualized as a unit sphere with an axis corresponding to the current direction of motion. As the vehicle and the corresponding set of points move, the position of the axis changes with respect to the sphere. This processing works well with temporal filters since there are constraints on how quickly a vehicle can change its direction of motion. This can also be used to determine if a vehicle is rotating with respect to an axis contained within the vehicle. This is indicated by areas of the image which show differences over time, but for which no clear axis of translation can be determined.

This tracking algorithm is based on the strong geometric constraints on image motion in the case of translational motion (radial motion of image features from a focus of expansion, determined by the intersection of the direction of translation with the imaging surface) [15]. The algorithm evaluates an error measure, which associates with a potential axis of translation the quality of feature displacements along the corresponding radial flow paths. This error measure is evaluated by searching over a unit sphere that describes all potential directions of translation. It is possible to determine the direction of translation to within a few degrees in small image areas, using only a few features.

If there is an instantiated 3D road model, and a rough estimate of the position of the vehicle along the road has been established, the tangent information associated with the road model can be used to initialize the search for the axis of translation. If there is an instantiated vehicle model, it restricts the features that the local translational tracker uses.

Feature Extraction

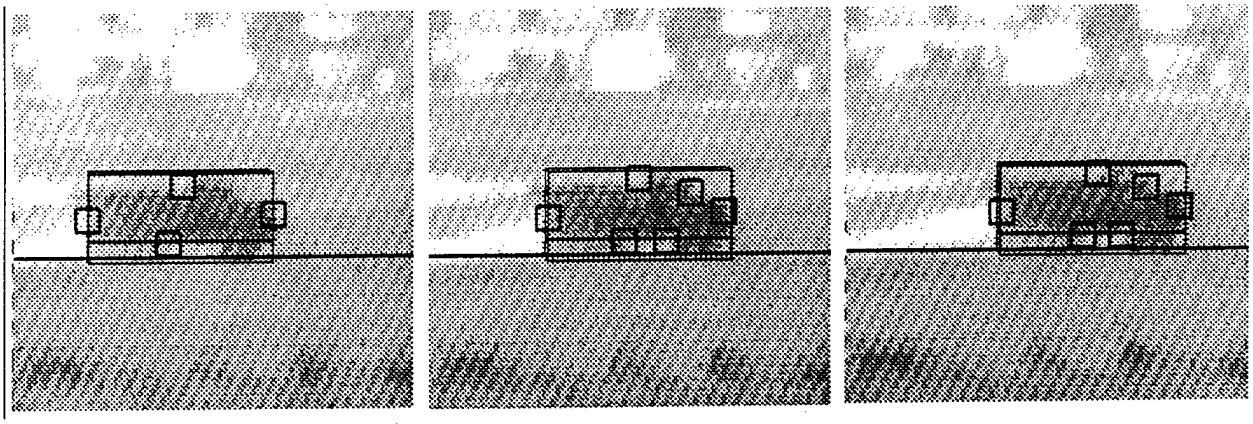

Figure 5.5. Extracted features with a superimposed 3D vehicle model.

The local translation-based tracker requires features that can be matched in successive images. The types of features used are conventional masks of image pixels, extracted from distinct areas of the image. In the examples shown in this chapter, the masks are 5x5 pixel arrays. Normalized correlation is used to determine the similarity of extracted features. This is used in measuring feature distinctiveness and in evaluating the matches of extracted features along the radial flow determined by a possible axis of translation. Since the radial flow lines do not necessarily pass through the center of the image pixel arrays, bilinear interpolation is used for matching features (see Figure 5.3). This allows the extraction of masks and performs correlation at a continuous range of locations, rather than just at the discrete pixel positions, and results in more accurate correlation values.

The distinctiveness of a feature is 1-minus the best correlation value obtained when the feature is correlated with its immediately neighboring areas. Good features are selected by finding the local maxima in the values of the distinctiveness measure over an image. We constrain the neighborhoods over which the features are selected to areas that contain large intensity discontinuities, determined by extracting zero-crossings. The area of feature extraction is further constrained by the output of the difference tracker or an instantiated vehicle and road model. The distinctiveness measure is then applied only to these restricted areas in an image. This generally results in the extraction of areas of high curvature along the zero-crossing contours. In addition, as a vehicle is tracked over a sequence of images, this processing is continually re-applied to find features in addition to those that have matched successfully. These can correspond to new features due to occlusions or changes in observable detail as a vehicle moves in depth.

Determining the Direction of Translation

Features in image sequences will move along radial lines defined by the focus of expansion (FOE) during translational motion. The FOE is determined by intersecting the direction of translation with the imaging surface (where the direction of translation emanates from the focal point of the camera). Using this geometric relationship, the displacement paths of all image features can be determined for a potential direction of translation. To evaluate a potential direction of translation, one searches for each feature along the appropriate image displacement paths. The error measure used to evaluate this potential direction of translation is determined by summing the best matches for each of the features.

To search for the direction of translation, a unit sphere centered at the focal point of the camera is used. Any vector that has its initial point at the camera's focal point and its terminal point resting on the surface of the sphere is a potential direction of translation. The search procedure is defined with respect to this sphere instead of the potential positions of the FOE in the image plane. This is because the sphere is a bounded surface, which makes uniform global sampling of the error measure feasible. When the image plane is used directly, the resolution in the position of the translational direction varies.

The initial search process consists of two phases: an initial global sampling of the sphere, followed by a local search for the maximum value. The local search begins at the position of the maximum value as determined by the global sampling. The local search process recursively searches the area of the current maximum. The step size of the local search processes is reduced until it is at the desired resolution to determine the direction of translation. Figure 5.6 shows a sequence of tessellated spheres, along with their potential directions of translation. Once a direction of motion has been established, it will tend to change smoothly and gradient-based techniques to track the axis of translation for successive images can then be used. In addition, if there is an oriented vehicle model or a road model segment, the search for the translational axis is constrained to limited areas of the sphere.

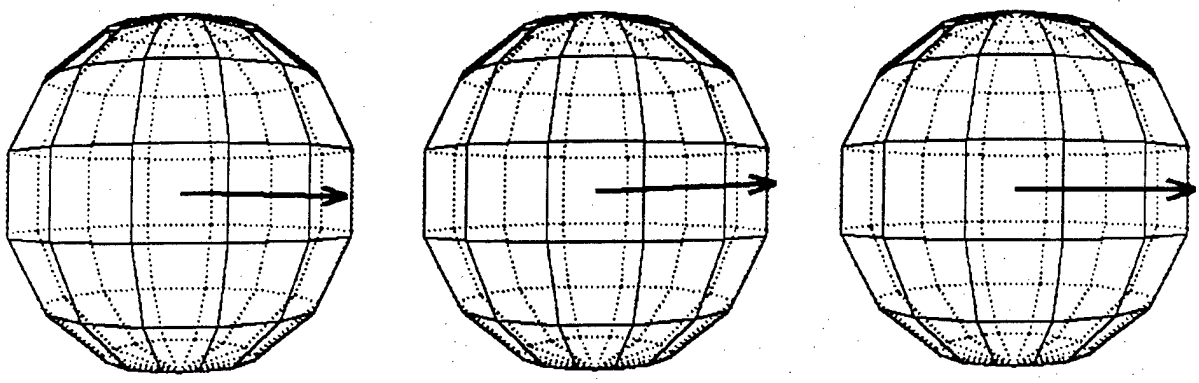

Figure 5.6. Translational motion spheres corresponding to the image sequence.

5.4.3 Planar Tracker

Often the motion of a vehicle is restricted to a plane determined by the local road or surface orientation. In this case, the geometry of planar perspective makes it possible to associate three-dimensional information with extracted image features if they are contained in the area of the planar patch. In addition, the directions of motion are constrained to be parallel to this plane, so the possible directions of motion for the local translation-based tracker are restricted to a circle on the unit sphere whose orientation is parallel to the plane of motion. This simplifies initialization and also tracking of the axis of translation over time.

There is another useful constraint associated with planar motion that may not be immediately apparent. In this case, an environmental displacement vector v must be perpendicular to the normal of the plane of motion. v also lies in the plane determined by its corresponding image displacement and the focal point of the camera. The direction of environmental motion can be determined by intersecting these planes. This is useful for tracking planar motion without the constraints supplied by a road model.

5.4.4 Feature Extraction from a Model

When the vehicle model is instantiated, it constrains segmentation and feature extraction procedures to a limited image area. In addition to the feature and zero-crossing extraction described above, a histogram-based segmentation is used to determine potential vehicle features.

An instantiated vehicle model can also constrain the places to search for detailed features corresponding to portions of the vehicle, which can be tracked. A particular problem we have found is that it is necessary to have a large image area to get clear views of the features to be matched to the model. Images of the vehicle will need to be larger to begin finding detailed features, such as headlights, bumpers, and so forth. Such images could perhaps be obtained by using one of the trackers to direct a zoom camera to follow a moving vehicle. Currently, we use the interest operator described for the translational tracker to match extracted features to a vehicle model for each successive image. If extracted features are near previously extracted features that

have successfully matched, they are discarded. Otherwise, they are associated with the instantiated vehicle model.

5.5 User Interface and Model Instantiation

An important facility in the user interface is a conventional depth buffer used for hidden surface removal, which has been modified to have pointers, ordered by depth, to all the objects in the world that project onto a given pixel in the image. Thus, when the human "touches" a pixel in the image from the telerobot, all the objects in the world model that project onto that pixel can be accessed. This is called an augmented depth buffer.

The user interface enables the human to place objects into the world model in several ways. The objects can be accessed and manipulated via their three-dimensional attributes with respect to a coordinate system linked to the world model. This looks like back projecting a 3D cartoon of the object onto the image. When it has been positioned as desired, the different components of the object can be placed in the augmented depth buffer associated with the image. In this way, the projected attributes of the instantiated object can access the actual image or the results of image-processing routines. The user can **burn-in** attributes when he instantiates an object. Burning-in means that the attributes can not be changed. This often involves constraining a particular feature to lie along a given ray of projection. Another technique is for the user to draw the specified object directly on the sensory input and then indicate its attributes. An example of this is interactively segmenting an image into different types of terrain patches and pointing out that different edges correspond to terrain feature discontinuity.

5.6 Processing Example

An example of this processing is shown in Figures 5.3 - 5.6. Figure 5.4 shows a sequence of images obtained with a video camera viewing a road scene. In Figure 5.3, a human has interactively positioned a generic vehicle model with respect to the road and has begun to "drive" the model vehicle through three-dimensions while using the back-projection of the vehicle as a three-dimensional cursor. Note the center segments of the road being laid down behind the vehicle. This establishes a 2D road mask and also an initial set of connected 3D road segments to constrain later processing. Figure 5.4 shows connected regions of image differences moving in a consistent direction with respect to the user instantiated road model. These correspond to the front and back of a vehicle. Since orientation is known along the road and the road model has been scaled relative to the generic road model, it is possible to use these areas to instantiate a three-dimensional vehicle model. Figure 5.5 shows interesting points which have been extracted in the corresponding areas determined by the vehicle model. These features are then used by the translation tracker to refine the estimate of vehicle and road orientation. The determined successive directions of translation are shown in Figure 5.6.

5.7 Future work

Current work involves extending many of the components described here. In particular:

- Extending the number and complexity of the models that are used. This includes developing an explicit inheritance hierarchy of models for different types of vehicles and terrain patches.
- Extending the user interface to use a wide range of interactive devices, such as a data glove and other three-dimensional positioning devices.

- Using more involved perceptual processing, especially for the segmentation that is sensitive to the material properties of objects and for the extraction of features to match directly to models.
- Using multiple cameras from different points of view with respect to the same scene. This will especially stress the importance of sensor calibration.

Chapter 6

Shape and Motion from Linear Features

6.1 Introduction

Chapter 6 introduces a technique for extracting structure and motion using directionally selective matches between linear features. A world-centered coordinate system is used to make these computations without the intermediate calculation of depth. In order to constrain the possible structure and motion configurations, we assume that the three-dimensional direction of gravity relative to each image frame is known. The direction of gravity, along with the directionally selective linear feature matches, forms a set of quadratic equations that can be used to determine structure and motion.

The extraction of environmental structure and motion from a sequence of two-dimensional images is a common problem in computer vision. Typically solutions to this problem are expressed in camera-centered coordinate systems where environmental geometry is specified by the depth along an image feature's ray of projection. Unfortunately, parameters computed from this camera-centered representation are dependent upon the depth of environmental features. This leads to erroneous results for objects located far from the camera.

The recently introduced *factorization method* [27, 28, 3] has attempted to overcome the disadvantages associated with a camera-centered representation. This method uses a world-centered coordinate system, along with an orthogonal projection assumption, in order to compute shape and motion without the intermediate calculation of depth. A matrix of image measurements is constructed by making point correspondences between image frames. The matrix is then factored into a shape matrix and a motion matrix using Singular Value Decomposition.

One problem with the factorization method is that it relies upon accurate point correspondences between image frames. This chapter introduces a method of extracting shape and motion from directionally selective linear feature correspondences. This line-based algorithm is capable of reconstructing shape and motion without computing depth as an intermediate step. In addition to the orthogonality assumption, it is assumed that the three-dimensional direction of gravity is known relative to each image in a motion sequence.

The algorithm begins by searching for the orientation of one of the lines in the environment. This is a one-dimensional search over 180°, constrained by the projection of the line on one of the image planes. Each candidate line orientation, along with the position of gravity, forms a set of quadratic equations, which constrain all the other lines as well as the rotation between image frames. An error measure is computed from the derived line orientations and is used to evaluate each shape and motion configuration. Once the line orientations and parameters of rotation have been derived, the relative positions of the lines can also be computed from simple linear equations.

The remainder of this section introduces the notation used throughout this chapter. Section 6.2 shows how to derive line orientation and camera rotation from a sequence of two-dimensional

images. Section 6.3 presents a set of linear equations that can be used to solve for the relative line positions. The algorithms presented in the paper are applied to synthetic data, and the results are presented in Section 6.4. Finally, concluding remarks are given in Section 6.5.

6.1.1 Notation

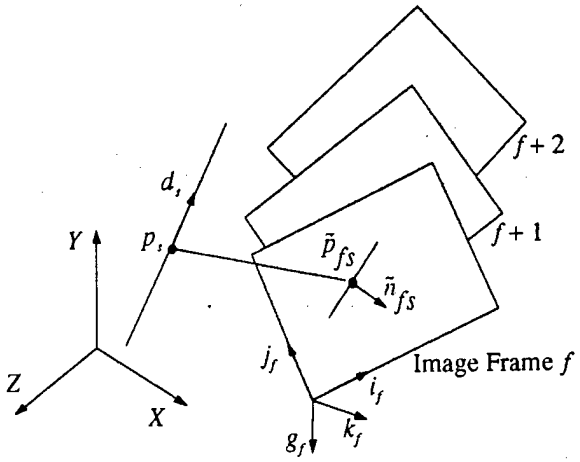

Figure 6.1. Coordinate systems.

The notation used throughout this paper is shown in Figure 6.1. An image frame at time f is delineated by unit vectors i_f, j_f, and k_f. A three-dimensional environmental line is represented by a unit vector d_s specifying the line direction, and a point on the line p_s. Line (d_s, p_s) is projected orthographically onto image frame f. The direction of the projected line is represented by its unit normal $\tilde{n}_{fs}$. $\tilde{p}_{fs}$ refers to the projection of p_s. The direction of gravity will be referred to as g_f. The two-dimensional parameters $\tilde{n}_{fs}$ and $\tilde{p}_{fs}$ as well as the three-dimensional parameter g_f are all expressed in the coordinate system of image frame f. All other parameters are specified relative to the world coordinate system. When $\tilde{n}_{fs}$ is specified in the world coordinate system, it will be referred to as n_{fs}.

In the following section, a method of solving for the line orientations d_s, as well as the parameters of rotation i_f, j_f, and k_f is presented. Section 6.3 shows how these initial quantities can be used to fix the relative positions of the lines within the world coordinate system by solving for a point p_s on each line.

6.2 Line Orientation and Camera Rotation

In this section, a method of solving for the three-dimensional line orientations and parameters of rotation from a sequence of two-dimensional images is presented. The algorithm begins by

searching for the orientation of one of the lines. This is a one-dimensional search over 180°, constrained by the projection of the line on one of the image planes. Each candidate line orientation, along with the position of gravity, forms a set of quadratic equations that constrain all the other lines, as well as the rotation between image frames. An error measure is computed from the derived line orientations and used to evaluate each shape and motion configuration. Section 6.2.1 shows how to solve for the position within the world coordinate system of the line normals (n_{fs}) associated with the candidate line. Section 6.2.2 shows how the candidate normals can be used to solve for the normals to all the other visible lines. These line normals are then used in Section 6.2.3 to estimate the line orientations and camera rotations.

6.2.1 Candidate Line Normals

The algorithm begins by searching for the orientation of one of the lines. A candidate line is used to constrain the position of all the other three-dimensional lines so that a particular shape and motion arrangement can be evaluated. The first step in this process is to solve for the position in the world coordinate system of the candidate line's normals. Let d_1 be the candidate line. Since the line normals $\tilde{n}_{f1}$ were formed by orthographic projection, they must be perpendicular to the line d_1. Therefore, one constraint is that the vectors n_{f1} must lie within the plane perpendicular to d_1. An additional constraint is provided by the gravity vector g_f. The angle between $\tilde{n}_{f1}$ and g_f must be the same as the angle between n_{f1} and the direction of gravity in the world coordinate system (g_w). These two constraints can be used to solve for n_{f1}. Figure 6.2 shows the geometry of these two constraints. Each normal (n_{f1}) is determined by intersecting a plane with a circular cone. The plane is defined by d_1. The cone is constructed by rotating a vector about the direction of gravity at the appropriate angle. Since the origin of the cone lies within the plane, the intersection of the plane with the cone results in two lines. There are only two possible solutions since the normals are known to be unit vectors.

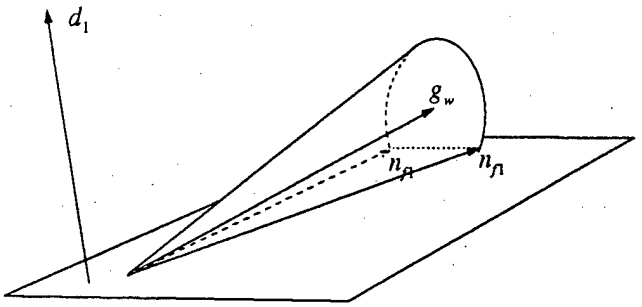

Figure 6.2. Normals are determined by intersecting a plane with a circular cone.

The constraints described above will now be examined in more detail. As stated earlier, the direction of gravity g_f relative to the line normals $\tilde{n}_{f1}$ is known. This results in the following relationship

$$n_{f1} \cdot g_w = \tilde{n}_{f1} \cdot g_f \tag{6.1}$$

where g_w is the direction of gravity in the world coordinate system. Letting g_w = (0, -1, 0), we can simplify Equation 6.1

$$n_{f1_y} = -\tilde{n}_{f1} \cdot g_f \tag{6.2}$$

In addition to the angle constraint, we know that n_{f1} lies within the plane defined by d_1. This constraint is expressed as

$$n_{f1} \cdot d_1 = 0 \tag{6.3}$$

Finally, we know that the magnitude of each normal vector (n_{f1}) equals one

$$\|n_{f1}\| = 1 \tag{6.4}$$

Equations 6.2, 6.3 and 6.4 can be combined into a single quadratic equation, resulting in two feasible solutions for each normal vector.

6.2.2 Additional Line Normals

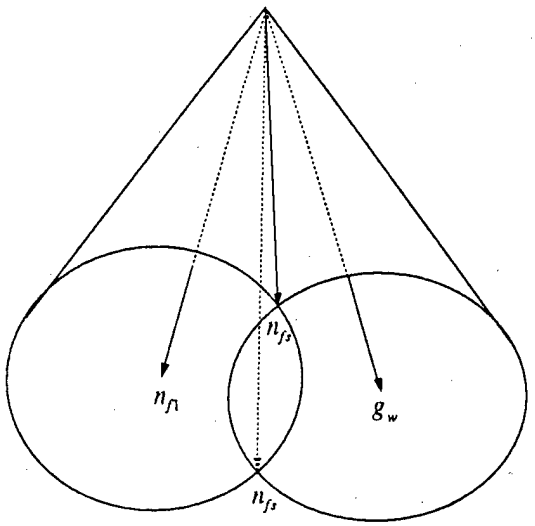

Figure 6.3. Normals are determined by intersecting two circular cones.

The next step in the extraction of line orientation and rotation is to solve for the position within the world coordinate system of the rest of the line normals. This is accomplished by using the candidate line normals. The idea is essentially the same as in the previous section. Two constraints can be formulated from the given geometry. The first constraint is given by the gravity vector g_f, and is identical to the constraint presented in the previous section. The angle between $\tilde{n}_{fs}$ and g_f must be the same as the angle between n_{fs} and the direction of gravity in the world coordinate system. The second constraint is that the angle between an image normal vector $\tilde{n}_{fs}$ and the candidate image normal vector $\tilde{n}_{f1}$ must be the same as the angle between the associated world

coordinate normal vectors n_{fs} and n_{f1}. These two constraints can be used to solve for all the additional normal vectors n_{fs}. The constraints are shown geometrically in Figure 6.3. The solution for a normal vector n_{fs} is essentially the result of intersecting two circular cones. One cone is the result of rotating a vector about the direction of gravity. The other cone results from rotating a vector about the candidate normal vector n_{f1}. The intersection of two circular cones which share the same origin is two lines. Once again, the normals are known to be unit vectors, resulting in two solutions.

The following equations result from the above analysis. The constraint resulting from the gravity vector g_f is identical to the one presented in Section 6.2. Therefore, from Equation 6.2 we can write

$$n_{fs_y} = -\tilde{n}_{fs} \cdot g_f \tag{6.5}$$

The second constraint relates the line normals n_{fs} to the candidate line normals n_{f1} as follows

$$n_{fs} \cdot n_{f1} = \tilde{n}_{fs} \cdot \tilde{n}_{f1} \tag{6.6}$$

Finally, we know that the magnitude of each normal vector (n_{fs}) equals one

$$\|n_{fs}\| = 1 \tag{6.7}$$

Equations 6.5, 6.6 and 6.7 can be combined into a single quadratic equation, resulting in two feasible solutions for each normal vector.

6.2.3 Parameter Estimation

Once the normal vectors (n_{fs}) have been derived, the process of estimating the line orientations and rotational parameters is trivial. The line orientations (d_s) are easily estimated from their associated normals (n_{fs}) using the following equation:

$$d_s \cdot n_{fs} = 0 \tag{6.8}$$

d_s can be estimated with a minimum of two non-collinear normal vectors. When more vectors are available, d_s can be solved for by using a linear least-squares technique. The rotational parameters are also easily obtained from the normal vectors n_{fs}. Three linear equations can be formulated for the three rotational parameters i_f, j_f, and k_f

$$\begin{aligned} i_f \cdot n_{fs} &= \tilde{n}_{fs_x} \\ j_f \cdot n_{fs} &= \tilde{n}_{fs_y} \\ k_f \cdot n_{fs} &= 0 \end{aligned}$$

There are also additional constraints available. One of these constraints is that the vectors must be orthonormal

$$\begin{aligned} i_f &= j_f \times k_f \\ j_f &= k_f \times i_f \\ k_f &= i_f \times j_f \\ \|i_f\| &= \|j_f\| = \|k_f\| = 1 \end{aligned}$$

Additional constraints can be derived from the relationship between the rotational vectors and gravity as was done in Sections 6.2.1 and 6.2.2. These constraints are

$$i_f \cdot g_w = g_{f_x}$$
$$j_f \cdot g_w = g_{f_y}$$
$$k_f \cdot g_w = g_{f_z}$$

Of course, all of the equations presented above are not independent, and all are not necessary. Currently, the following subset of equations is used. Initially k_f is determined using a least squares formulation of

$$k_f \cdot n_{fs} = 0 \tag{6.9}$$

The technique presented in Section 6.2.1 is then used to solve for i_f with the following equations

$$i_f \cdot k_f = 0 \tag{6.10}$$
$$i_f \cdot g_w = g_{f_x} \tag{6.11}$$

Finally i_f and k_f are used to solve for j_f

$$j_f = k_f \times i_f \tag{6.12}$$

Equation 6.8 is used to solve for the line orientations d_s. Equations 6.9, 6.10, 6.11 and 6.12 are used to solve for the rotational parameters i_f, j_f, and k_f. The following section shows how to use these derived parameters to solve for the relative positions of the line segments, thus completing the spatial reconstruction.

6.3 Line Position

The final step in the line segment reconstruction is to solve for the line segment positions relative to the world coordinate system. Initial assumptions about the position of the image frames relative to the world coordinate system are made, allowing a simple linear solution to the problem. The position of each line is represented by a point p_s which is chosen arbitrarily. The world coordinate system will be positioned at the center of image frame 1. The points $\tilde{p}_{1s}$ are then chosen arbitrarily $\tilde{p}_{1s} = (x_s, y_s)$. We assume that all the image planes intersect along line d_1. This means that the position of each image plane is given by $p_1 + \alpha_f d_s$ where α_f is a parametric scale factor.

Each line position $p_s = (x_s, y_s, z_s)$ consists of one unknown z_s. The solution for z_s is trivial. Each point p_s is constrained to lie within the planes perpendicular to n_{fs}. These planes are positioned by choosing some arbitrary point on the projection of each line, and then determining the position of that point within the world coordinate system. Let q be the point in world coordinates

$$q = p_1 = [i_f j_f] \cdot (\tilde{p}_{fs} - \tilde{p}_{f1}) \tag{6.13}$$

The equation of the plane is then written as

$$\begin{aligned} & n_{fs_x}(x_s - q_x - \alpha_f d_{s_x}) + \\ & n_{fs_y}(y_s - q_y - \alpha_f d_{s_y}) + \\ & n_{fs_z}(z_s - q_z - \alpha_f d_{s_z}) = 0 \end{aligned} \tag{6.14}$$

The two unknowns in this equation are z_s and α_f. α_f can be removed from the equation, and a least squares solution can be found for z_s.

6.4 Results

The algorithm presented in this chapter was implemented and tested on several sequences of synthetic data. The first and last frames from a 20-image sequence are shown in Figure 6.4.

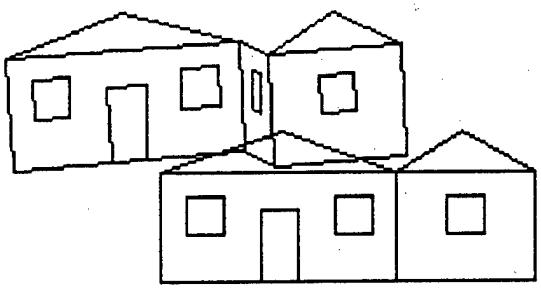

Figure 6.4. The first and last frames from a 20-image sequence.

Figure 6.5 shows 10 frames from the sequence (every other frame is displayed). This data was produced by random rotations and translations.

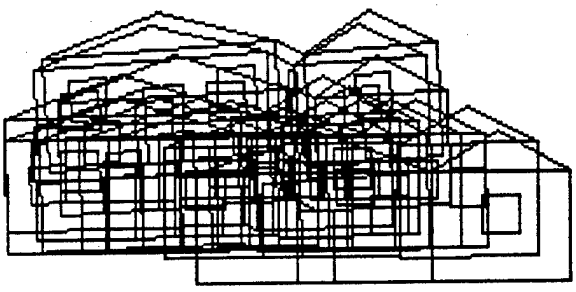

Figure 6.5. Ten image frames from a 20-image sequence.

The rotational parameters i_f and j_f associated with this sequence of motion are shown in Figures 6.6 and 6.7. The correct rotational values are displayed as solid lines, and the derived values are displayed as dotted lines. All errors are the result of perspective projection. Notice that the Y-component of i_f is errorless. This is because this component is derived from the relationship between the image frames and the gravity vector (g_f), as shown in Equation 6.11. Thus, the Y-component is unaffected by the perspective projection errors.

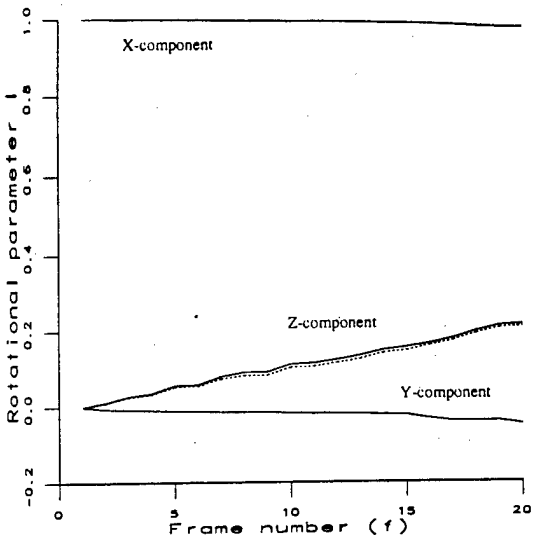

Figure 6.6. The components of i_f for a 20-frame sequence. The correct values are shown with solid lines, and the derived values are shown with dotted lines.

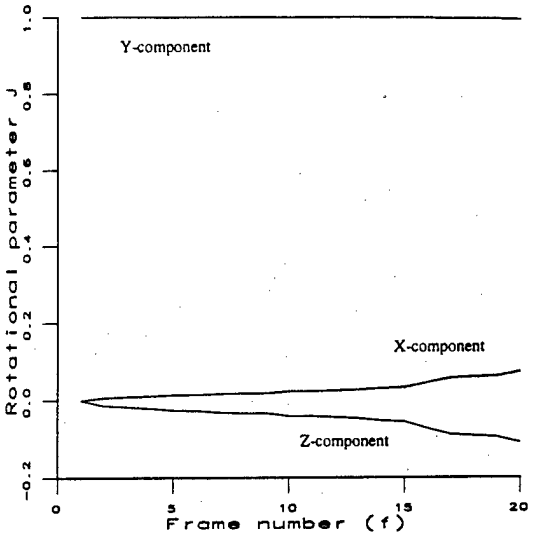

Figure 6.7. The components of j_f for a 20-frame sequence. The correct values are shown with solid lines, and the derived values are shown with dotted lines.

The derived line orientations and parameters of rotation were then used to reconstruct the line positions as discussed in Section 6.3. A top view of the original data is shown in Figure 6.8.

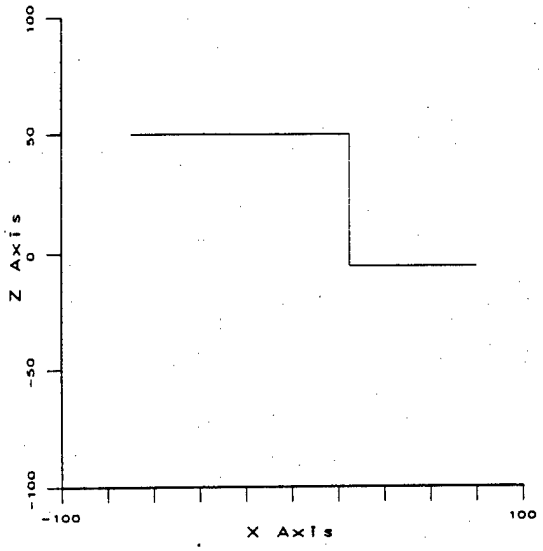

Figure 6.8. Top view of the house data.

The reconstructed data is shown in Figure 6.9. Once again the errors are the result of perspective projection.

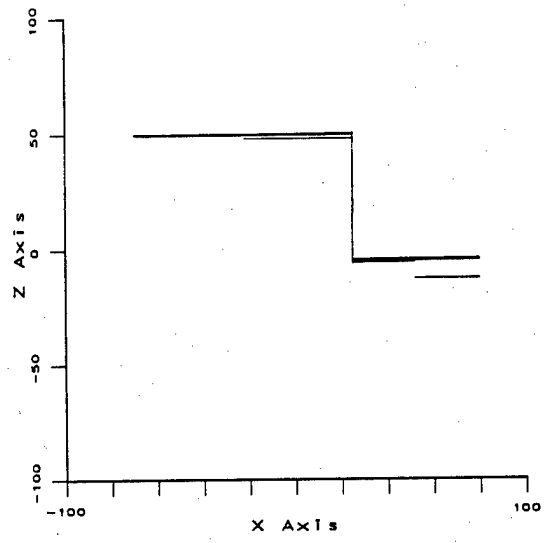

Figure 6.9. Top view of the reconstructed house.

6.5 Future Work

The technique presented in this chapter is an early attempt at constructing linear feature-based, depth-independent motion algorithms. The work has only been tested on synthetic data, and it is

not clear what effect perspective projection and other forms of noise will have. However, since the formulation involves linear least squares estimation, it appears that it will be robust. The ability to deal with occlusion is also straight-forward in this over-constrained system. Occluded line normals (n_{fs}) are null vectors and therefore have no effect on the least squares solution. Notice that the first frame shown in Figure 6.4 contains occluded lines.

One drawback of this method is that the three-dimensional direction of gravity is required. This measurement can be provided by a gravity sensor, but we would like to relax this restriction. One way to remove the gravity vector from the algorithm is to replace the direction of gravity with another consistent direction. For example, for an object that consistently moves in one direction (such as a vehicle), the gravity vector can be replaced by a vector specifying this direction (the forward vehicle direction).

There are several areas for future work:

- Test this algorithm on noisy data and, if necessary, develop a more robust formulation that will work well in the presence of errors, including the errors introduced from perspective projection.
- Test the algorithm on real image sequences.
- Integrate this rotation-based method with the translation-based method discussed in [15]. In this case, the gravity vector is replaced by a direction of translation vector. The integration of these two methods will probably be accomplished through temporal filtering using the Kalman Filter.

Chapter 7

Range Free Robot Navigation

7.1 Introduction

This chapter will summarize some recent algorithms developed for qualitative navigation, which are completely independent of range estimates to landmarks. Several distinctions that reflect more realistic application of qualitative navigation algorithms to real robots are introduced. These involve the extent to which landmarks can be identified from very different points of view (called the *distinctiveness* of landmarks); whether or not a compass is allowed; and distinctions between different types of compasses.

Qualitative Navigation [13, 12, 21] concerns spatial learning and path planning in the absence of a single global coordinate system for describing locations and the positions of landmarks. It is based on a multi-level representation of space, which, at its most abstract level, is based on topological properties that allow a robot to describe a location using the directions of visually salient patterns (with no associated range measurements) and then navigating using the occlusions that occur among them as a basic cue to control movement through the environment. An advantage is that the robot can use landmarks for which exact positions cannot be determined. Thus, if a robot sees a building in the distance, it may not know or be able to recognize the structure as a building or determine its exact position in space, but it can still incorporate this to form an effective spatial memory. This is actually quite intuitive. It is doubtful that animals navigate by detecting landmarks, determining ranges to them, and then storing everything in a single frame of reference [8]. The robot also removes the effects of incremental errors due to drift.

This work [21] in qualitative navigation developed while trying to produce basic navigation and recognition capabilities in an autonomous land vehicle. Initially, the project worked with a terrain representation based upon an *a priori* terrain grid, which describes terrain in terms of a regular square grid of features referenced with respect to a single global coordinate system. Several problems were discovered with such spatial representations. The grids would describe large patches of terrain by a set of numbers that corresponded to terrain features such as elevation and vegetation type. Unfortunately, the world consists of objects that are difficult to summarize by a single set of numbers. It is difficult to establish the exact three-dimensional position of a distant landmark, especially when using passive sensing. Thus, it is difficult to know where to attach landmarks to the underlying terrain representation when it uses a single, global coordinate system. Robots also have limited recognition capabilities in complex outdoor environments. They can see distinctive things in the world, and yet not know what or where they were. In fact, there are no assurances that robots can see the same object as being the same object from very different points of view.

Qualitative Navigation deals with these problems via a multi-level representation of spatial memory. The different levels are distinguished by what constitutes a landmark and by the **connectedness** of spatial memory, which refers to how, given one location, it is possible to

determine the position of another location. At the simplest level of spatial representation (**the Sensorimotor level**), a landmark consists of a perceptual event that can be used for sensory feedback to control guidance. The next level (**the Topological level**) is based upon noting and tracking stable perceptual events around the robot, but without associating any *range* information to these. This level is topological in the sense that there is no metric information associated with landmarks. A place is described by the set of visual patterns surrounding the robot. This description of a place is called a **viewframe**. Movement from place to place is determined when there is some change in the order of these patterns. The next level allows the association of potentially inexact range information with the visual patterns (**Local Coordinate systems**). At this level, viewframes can have associated range estimates with the detected visual patterns and the localization of one place to another was inexact. The final level (**Global Coordinate System**) assumes that we have exact three-dimensional information for all landmarks. In [21], it was found that by working at the level of a viewframe-based representation, the problems faced when working with a single global coordinate system were drastically simplified.

This chapter describes qualitative navigation algorithms that work completely at the topological level, dealing with landmarks for which there are no range estimates. In addition, several distinctions for qualitative navigation algorithms are introduced. One type of distinction concerns landmarks. This research considers two basic types: **distinct landmarks** that can always be recognized as the same from wherever they are seen and **non-distinct landmarks** that may not be recognized as being the same when seen from different points of view. One can assume that once landmarks are seen, they can be tracked over time until they disappear. The other distinction involves whether or not the navigation algorithms use a compass to yield a fixed direction. Two different types of compasses were distinguished. The direction associated with a **local compass** can change from place to place but, at a given place, it will always point in the same direction. An example is a compass that is effected by fixed magnetic influences at different locations. The local compass can also be a very strong landmark that is visible from a wide set of views. A **global compass** will always point in the same direction regardless of where the robot is located. These distinctions can be expressed as a table corresponding to the different types of topological navigation algorithms we have developed:

Topological Qualitative Navigation Algorithms

	Compass	No Compass
distinct landmarks	Very Good	Good
non-distinct landmarks	Good	Difficult!

For example, consider qualitative navigation without a compass and identical, non-distinct landmarks. As one might expect, this is very difficult and depends critically on matching viewframes based exclusively upon the angular orientations of landmarks. More practical algorithms are those that are based upon using a local compass and a limited number of distinct landmarks. This corresponds to a freely navigating robot that can build maps and navigate using simple visual features, such as colored regions and edges aligned with gravity, as landmarks.

The remainder of this chapter will describe the basic memory organization used for qualitative navigation, and then present different navigation algorithms.

7.2 Organization of Spatial Memory and Navigation Behaviors

7.2.1 Landmarks

This research distinguishs between types of landmarks to reflect different recognition capabilities in robots. A *distinct landmark* is one that can be recognized as being the same from all points of view. Distinct landmarks require considerable recognition capabilities for a robot owing to the variable appearance of landmarks from different points of view. A *nondistinct landmark* is one that may not be recognized as being the same from different points of view. The assumption is made that once a nondistinct landmark is seen, it can be tracked over time until it disappears. A nondistinct landmark is not necessarily described as a particular object in the world, but can be described as a simple visual pattern, such as a colored region of a particular shape or a set of edges aligned with gravity. Such descriptions of landmarks will tend not to be unique.

A general finding of the algorithms described here is that the more distinct landmarks there are, the more easily a robot can find shortcuts and novel paths between locations. The more indistinct landmarks there are, determining position depends on recognizing the distribution of landmarks surrounding a robot. In this case, the robot will tend to stay close to established paths that it determines during explorations. It is possible for a robot to determine novel paths between locations with nondistinct landmarks, but it requires significant exploration to determine that a landmark is the same from many different points of view.

7.2.2 Viewframes

A **viewframe** contains the set of visible landmarks surrounding a robot at a given location with their corresponding orientations and other attributes describing the individual landmarks (such as color, visible height, contrast, etc.). Viewframes are a one-dimensional sequence of landmarks (the direction of gravity is used to reduce the two-dimensional images surrounding the robot to a one-dimensional sequence). An example viewframe V is shown in Figure 7.1. This viewframe uses compass information and is then represented as

[Viewframe Identifier: V
Landmarks:
$[[lid_A; AttributesA], \alpha_A]$
$[[lid_B; AttributesB], \alpha_B]$
$[[lid_C; AttributesC], \alpha_C]$
Robot's heading: $\alpha_h]$

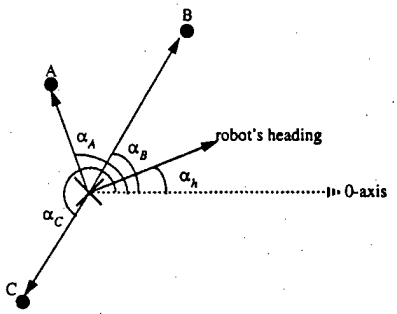

Figure 7.1. Viewframe Representation with a Compass.

When a viewframe is extracted without a compass, there is no associated 0-axis to describe a fixed direction. The relative orientation of landmarks is then represented by the angle difference between successive landmarks. The same viewframe in Figure 7.1 is represented (shown in figure 7.2) as

[Viewframe Identifier: & V
Landmarks:
$[[lid_A; AttributesA], \theta_A]$
$[[lid_B; AttributesB], \theta_B]$
$[[lid_C; AttributesC], \theta_C]$
Robot's heading: $(\theta_h, B)]$

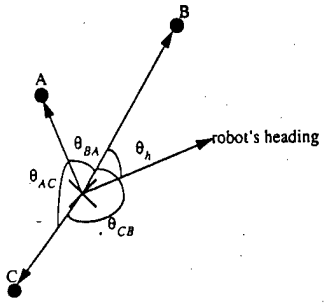

Figure 7.2. Viewframe Without a Compass.

For the viewframe in Figure 7.1 and Figure 7.2, lid_A, lid_B, lid_C are the *local identifiers* for visible landmarks A, B, C. The **local identifier** is a name or abstraction of the attributes of a landmark that is tied to a specific viewframe. Note that a landmark with the same local identifier in different viewframes can have different image attributes depending upon the viewframe it is contained in. A distinct landmark that can be recognized as being the same from very different points of view has a unique local identifier with respect to all viewframes and is called a **global identifier**. When a robot is exploring the environment, distinct landmarks will always be associated with a unique local identifier in all the viewframes that contain it. Nondistinct landmarks will have the same local identifiers in connected viewframes as long as the landmark is visible (or after landmark-unification – see below). When a landmark reappears or is disoccluded, it will have a new associated local identifier. This is similar to what can happen when an animal walks on two different paths without realizing that there is a common landmark between them. For each nondistinct landmark, there can be more than one local identifier for it in different viewframes.

When viewframes consist largely or totally of nondistinct landmarks, being able to access or recognize a particular viewframe is difficult (for example, a large red region can be a landmark in several different viewframes). For this reason, keys are also associated with viewframes that are used for recognizing viewframes by a hashing operation. There are a large number of different keys, such as the average height of landmarks, the average angle between landmarks, the number of landmarks, number of highest landmarks, number of landmarks for particular colors, variance of contrasts, variance of heights, variance of angles between landmarks, ratios of landmarks having different attributes, etc. These keys help to distinguish and match viewframes. If there is a local compass, many more types of keys are possible because it is possible to order the landmarks in the viewframe and compute keys based upon position in the viewframe. Each key has a limited number of values. Two viewframes are said to be *hash-matched* if they have the same key value for each key.

Keys are also useful for the efficiency of accessing viewframes. Suppose there are 10 keys, each key has 10 different values; therefore, there are 10^{10} equivalence classes. We build a hash table, making an entry for each possible value combination of all keys. To find a viewframe to match V, we first compute key values of V for all keys, then use the combination of those values as an index to the hash table to find the viewframe in the database. Since we have $O(1)$ number of keys, time to compute the key value is $O(1)$, time to search in the hash table is $O(1)$; therefore, the time complexity to find a viewframe to match V is reduced to $O(1)$.

7.2.3 Viewframe Extraction and Filtering

The extraction of a viewframe involves identifying landmarks surrounding the robot. These are then stored in different types of viewframes depending upon whether or not there is a compass and on the distinctiveness of the landmarks. It has also been found useful to compare a newly extracted viewframe to the previously extracted viewframe to determine if the newly extracted viewframe is different or novel enough to merit storing it in spatial memory. This process is called *viewframe-filtering* and has the effect of reducing the number of very similar or redundant viewframes that are stored in memory. Filtering is done by keeping track of changes in the values associated with the different keys. There is a threshold associated with allowable changes in the value for each key. If this is exceeded, then the viewframe is stored in spatial memory. For example, if the number of landmarks changes drastically, it is necessary to then extract a viewframe in spatial memory. It may also be useful to have a function that weighs the changes in the different keys to determine whether a viewframe is novel enough to be extracted.

7.2.4 ViewFrame Matching

Viewframe matching is the process that determines the similarity of two viewframes. A two-level matching processing is used. The first level finds similar viewframes by hashing and then uses the number of landmarks with common local identifiers in both viewframes as a measure of similarity called *connectivity*. First-level connectivity between two viewframes is defined as:

$$con(V_1, V_2) = \frac{\left\| Local\_ids(V_1) \bigcap Local\_ids(V_2) \right\|}{\left\| Local\_ids(V_1) \bigcup Local\_ids(V_2) \right\|} \qquad (7.1)$$

Second-level viewframe matching compares the orientation (angle) difference between landmarks. For this level of matching, different thresholds for the maximum orientation difference for corresponding local identifiers in the viewframes are used.

7.2.5 Navigation Behaviors

The navigation algorithms are based on a set of simple, visual tracking behaviors. **Viewframe crossing** is when the robot is positioned at a landmark and walks in the direction of the center of a viewframe which contain that landmark. Viewframe crossing will generally depend on a local compass, which is valid within the extracted viewframe. **Viewframe back-matching** (also called **landmark unification**) involves recognizing that landmarks in different viewframes are the same and their local identifiers are unified. This happens when a robot visits the same place along separate paths. **Landmark circling** is when a robot circles around a known landmark. It is a way of searching for surrounding landmarks when no nearby landmarks are distinguished or visible. The robot can spiral towards or away from the landmark (until the landmark is no longer visible). **Landmark targeting** is for walking towards a visible landmark. **LBP crossing** is when a robot crosses a Linear Pair Boundary defined by two landmarks. The crossing can occur

on either side or through the center of the LPB between the two landmarks. **LBP alignment** is when the robot travels along the LPB boundary defined by two landmarks. **Random walking** randomly selects a visible landmark, walks to it, and then repeats. An alternative version walks straight for some distance, changes direction, and then repeats. In **Novelty walking**, a robot walks to optimize the changes in the keys used for viewframe filtering. The effect is to go someplace where it is as different as possible from where you currently are.

7.2.6 Spatial Memory

Spatial memory consists of three inter-related databases: the viewframe database (**V-DB**), the path database (**P-DB**), and the landmark database (**L-DB**) (see Figure 7.3). The landmark database contains descriptions of landmarks that a robot has seen. It is possible for the same physical landmark to occur several different times in the landmark database, because it may not have been identified as being the same from different views. The viewframe database contains viewframes that describe the visible landmarks surrounding a robot at a given location (this is described in more detail shortly). The path database consists of connected sequences of viewframes that a robot determines while exploring the environment. Database Storage algorithm:

Step 1 Extract *VF* and filter against previously extracted *VF*.

Step 2 Compare *VF* with other viewframe in VF-DB by some viewframe matching mechanism.

Step 3 If *NOT* matched for *VF*, add *VF* to V-DB, add pointer to *VF* into each landmark entry with the same local identifier in L-DB; or return the pointer to *VF* in V-DB.

Step 4 Add pointers to *VF* into current path's viewframe sequence in P-DB.

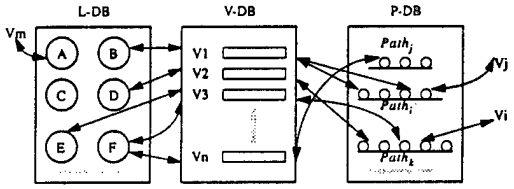

Figure 7.3. Memory Architecture.

7.2.7 Qualitative Navigation Simulator

Different qualitative navigation schemes using the simulators shown in Figure 7.4 and Figure 7.5 (for exploring indoor navigation) have been explored. Each contains four subwindows. The upper-left subwindow is a Unix shell. The lower-left subwindow has controls for setting such things as the density of landmarks, the range of visibility, the number of globally distinct landmarks, and for selecting different navigation modes. The upper-right subwindow shows the 360 degrees of view from the robot at a given location. The lower-right subwindow shows a top-down view of the navigation world. In Figure 7.4, the circle shows current viewframes containing

landmarks displayed in the upper-right subwindow. The line in the circle shows the robot's heading. Distinct landmarks are numbered and nondistinct landmarks are not numbered, but can appear as having different colors and intensities.

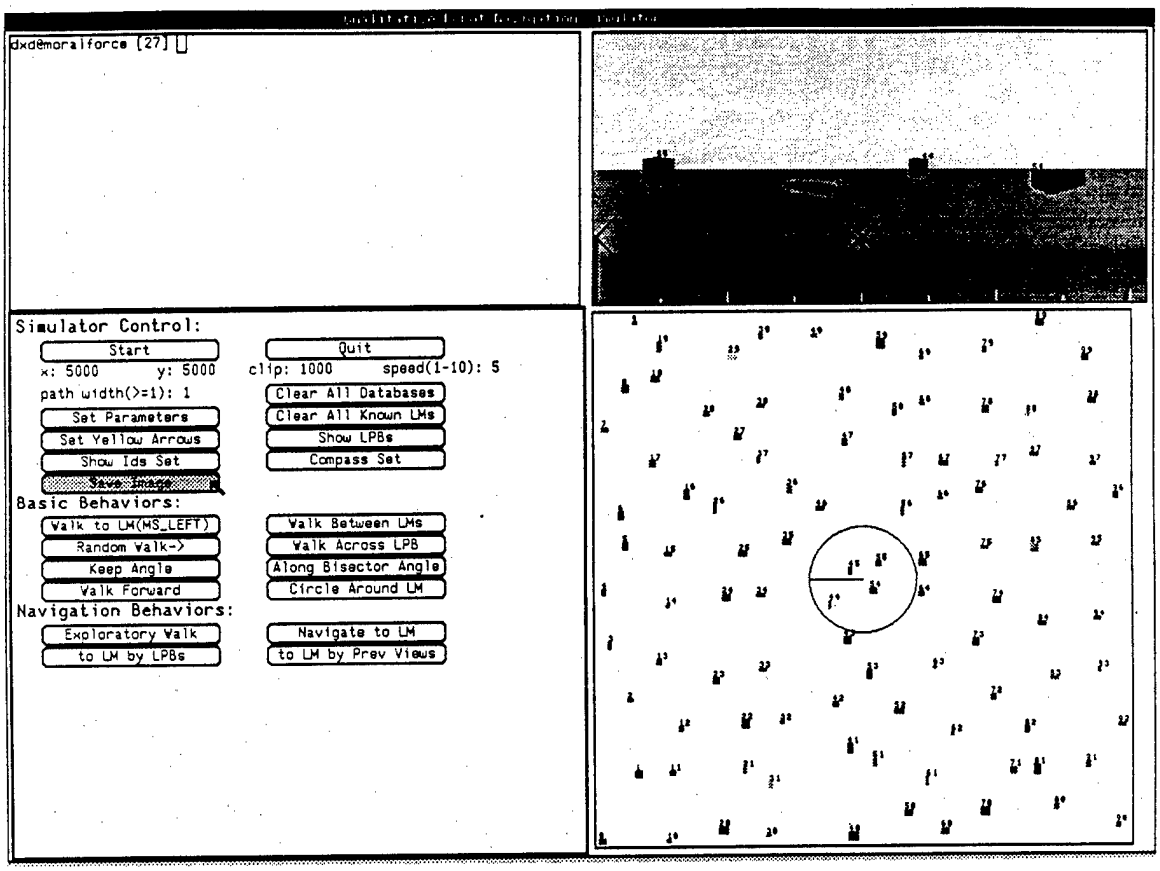

Figure 7.4. Qualitative Navigation Simulator.

In the simulator in Figure 7.5, the assumption is made that limitations on sight are only caused by occlusion. The current viewframe is shown as the set of radiating lines from the robot's current position to each of the visible landmarks. The current viewframe is displayed as a sequence of landmarks in the upper right-hand window.

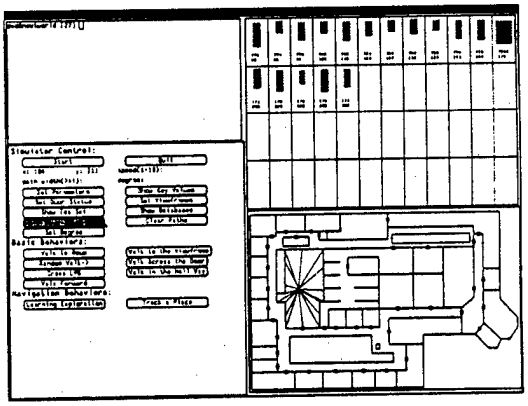

Figure 7.5. Simulator for Indoor Robot with displayed viewframe.

7.3 Navigation Using A Local Compass With A Variable Percentage Of Distinct Landmarks

This algorithm is intended to work with a variable number of distinct landmarks, ranging from completely nondistinct landmarks to completely distinct landmarks. The nondistinct case would correspond to walking through a world full of identical landmarks with a compass. When the number of distinct landmarks increases, the efficiency of the path planning improves.

Navigation using this algorithm is shown in Figures 7.6 through 7.8. The robot initially walks along two separate paths which, form a Ψ-like shape shown by the solid thin lines. The robot is first at the upper-middle part of the Ψ and walks to the middle-lower part. It is then relocated to the upper-left corner and walks to the upper-right corner along a curved path. As it walks along these paths, it keeps track of landmarks and stores extracted viewframes in the different system databases. It then has the task of going from the upper-right corner of Ψ to the upper-left corner of Ψ. To do this, the robot can either follow the long curved path that it originally followed, or it can find a short-cut directly between them. The key result is that as the number of distinct landmarks increases, the robot is able to find increasingly more direct paths between locations. With more nondistinct landmarks, navigation involves staying close to paths that have been previously followed. Shortcuts are possible when common landmarks between paths are found.

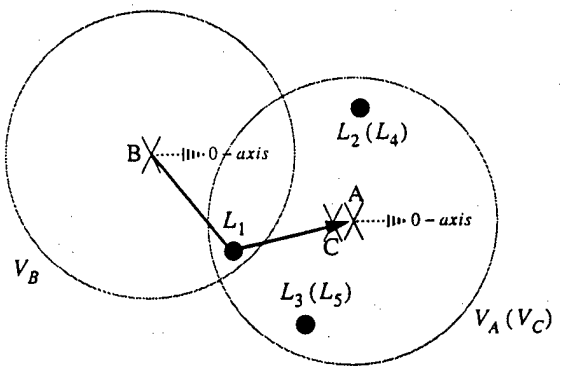

Figure 7.6. 100% Nondistinct Landmarks. Path Planning (Solid Thick Path) from Upper-Right Corner to Upper-Left Corner of the Ψ.

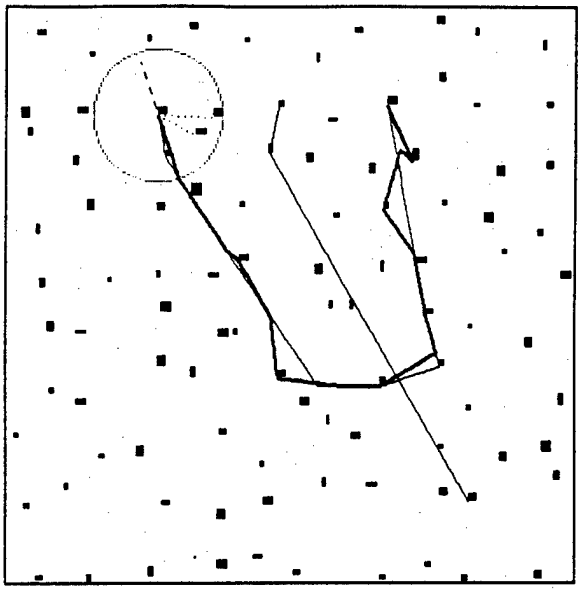

Figure 7.7. 75% Nondistinct Landmarks. Path Planning (Solid Thick Path) from Upper-Right Corner to Upper-Left Corner of the Ψ.

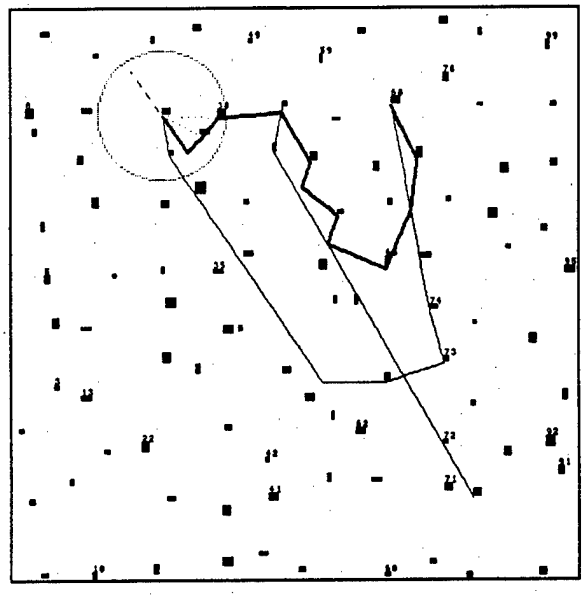

Figure 7.8. 50% to 0% Nondistinct Landmarks. Path Planning (Solid Thick Path) from Upper-Right Corner to Upper-Left Corner of the Ψ.

The algorithm utilizes viewframe crossing and viewframe back-matching. In **Viewframe crossing**, a robot walks from a landmark towards the center of a viewframe that contains that landmark. If the robot is at a landmark with local identifier L and orientation angle α in V, viewframe crossing is to walk in orientation angle $\alpha+\pi$ towards the center of V. Two

viewframes are said to be *connected or adjacent* if they have at least one local identifier in common. Navigation then involves finding a sequence of connected viewframes (V_o, ..., V_j, ..., V_n) with overlapping landmarks that are traversed by successive viewframe crossing.

Viewframe back-matching is used to determine that landmarks having different local identifiers in different viewframes actually are the same physical landmark. They can then be used to navigate from one viewframe to another and to form the basis of finding shortcuts when such common landmarks are recognized. During viewframe crossing to V_j, if the robot cannot find a *distinct* landmark in common between V_j and V_m ($m > j$ and $m < n$), it attempts viewframe back-matching to update local identifiers in the viewframe database. This is illustrated in Figure 7.9.

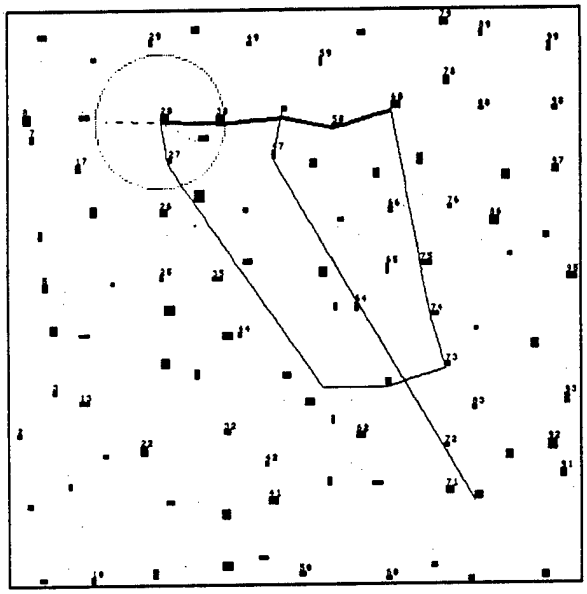

Figure 7.9. Viewframe Back-Matching with a Compass.

The robot is currently at V_B and has previously extracted V_A with local identifiers L_1, L_2, L_3 associated with the nondistinct landmarks. V_A and V_B have local identifier L_1 in common. The robot first goes to landmark L_1, then walks towards the center of V_A. While it is walking towards the center of V_A, it continues to extract viewframes and perform second-level viewframe matching (based upon angle and orientations of landmarks) with respect to V_A. When it extracts a viewframe at C (which is nearby A) with new local identifiers L_4 for L_2, L_5 for L_3, the robot matches V_C to V_A, updating the viewframe database by substituting L_3 into L_5, L_2 into L_4.

The algorithm has the following steps:

Goal: A landmark with local identifier *lcid* (tied to a specific viewframe) to go to.

Step 0 If *lcid* is in current viewframe, go directly to it and the algorithm terminates.

Step 1 Create a *virtual* viewframe V_n containing only the goal *lcid* with an unspecified orientation. Construct an N by N weight matrix W (N is current total number of viewframes plus one). For each pair of viewframes (V_i, V_j), including the *virtual*

viewframe V_n and current starting viewframe V_0, compute $con\ (V_i, V_j)$ by equation (1), if it is greater than a certain threshold (select 0), assign the weight matrix entry $W(i,j) = 1$; otherwise $W(i,j) = \infty$. With the weight matrix, we find a *least* sequence of connected viewframes V_0, V_1, ..., V_n by applying a shortest path algorithm [6]. Alternatively, if the total number of viewframes N is too great, a breadth-first tree search [6] is used from V_0 to find adjacent viewframes, such that a viewframe cannot appear twice in a path of the tree.

Step 2 If a sequence of *connected* viewframes are not found, stop. Otherwise the robot performs *viewframe crossing* and *viewframe back-matching* through V_0, V_1, ..., V_n. It walks to the landmark with common local identifier in both V_0 and V_1, where choice of distinct landmark has priority.

Step 2.1 If the robot is currently at landmark P of viewframe V_i(i is max), it *viewframe-crosses* towards the center of V_i, testing if current viewframe V_c is adjacent to V_m, i.e. $m \in [i+1, n]$ and m is max such that $con(V_c, V_m) > 0$; if m is found, which means a distinct landmark is found, then it changes the direction and walks to the landmark in *both* V_c and V_m. Otherwise, it performs *back-matching* to V_i; if no ambiguity occurs and the best match is found, the robot updates the local identifiers, i.e. it uses local identifiers in V_i to replace corresponding local identifiers with the same orientations in V_c as well as those local identifiers in V-DB; and then walks to a landmark with a common local identifier both in V_i and V_{i+1}.

Step 2.2 Repeat Step 2.1 until the goal is achieved, or failure due to ambiguity.

When all the landmarks are distinct, viewframe back-matching is unnecessary.

7.4 Navigation Without a Compass for Distinct Landmarks (No LPBs)

This algorithm assumes distinct landmarks, no compass and no LPBs (Landmark-Pair-Boundary), which is described in the next section.

The algorithm relies on **viewframe circling** to compensate for the lack of a compass. Figure 7.10 shows an example of navigation using this algorithm with the viewframes and paths from the previous figures. The exploration paths Ψ (solid thin lines) are generated in the same manner as in Figures 7.7, 7.8 and 7.9. The path determined by the robot is shown as a solid thick line from the upper-right corner of the Ψ to the upper-left corner of the Ψ. The result shows that the path is slightly longer than that with a compass (in Figure 7.9).

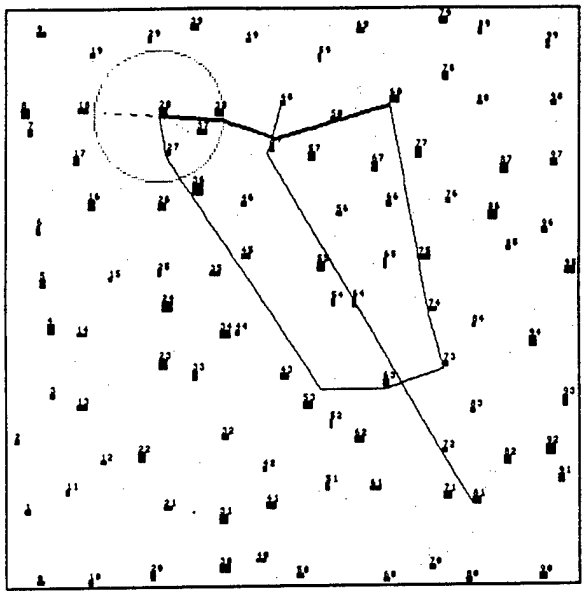

Figure 7.10. Path Planning from landmark 68 to 28 (thick path).

The processing example in Figures 7.11, 7.12, 7.13 and 7.14 shows some of the interesting characteristics of this algorithm. In Figure 7.11, the robot moves to Landmark 89 by an exploratory behavior to generate a path. One then wants the robot to walk between Landmark 89 and 64, so it essentially walks back to where it had been.

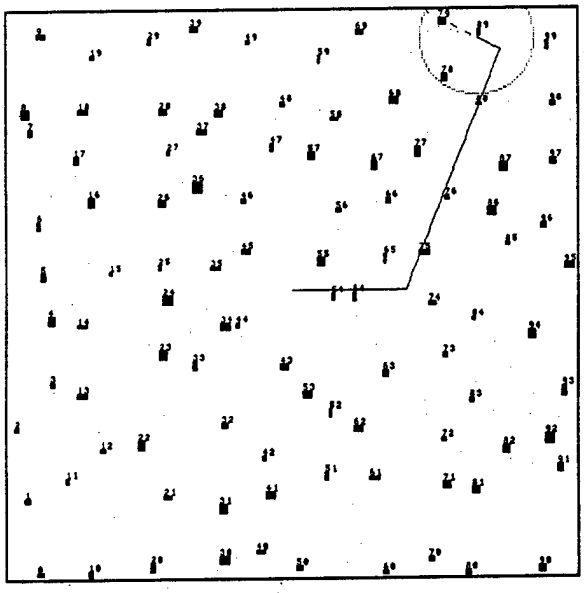

Figure 7.11. Exploration Path Generated by Basic Behaviors to landmark 89.

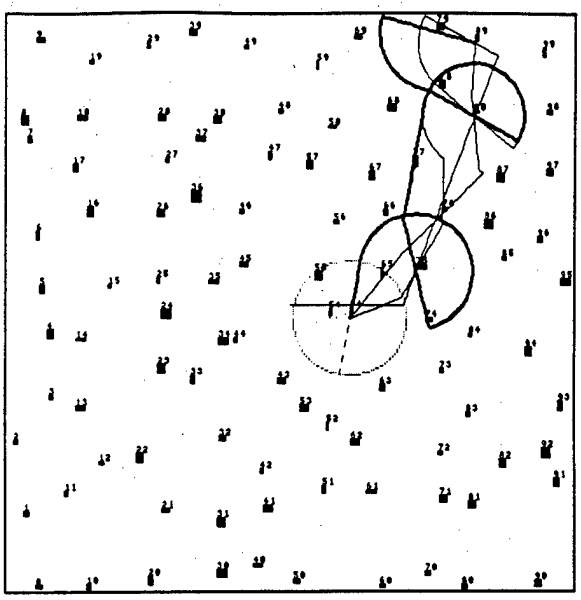

Figure 7.12. 3rd Time from landmark 89 to 64 (thick path).

Initially, the robot cannot find, due to limitations on its range of vision, most of the visible landmarks along its path (Figure 7.12). So it begins to circle around the current landmark to search for landmarks from the path it traversed. This continues until it returns to its origin. The circling behavior for finding landmarks is responsible for the indirect looking paths.

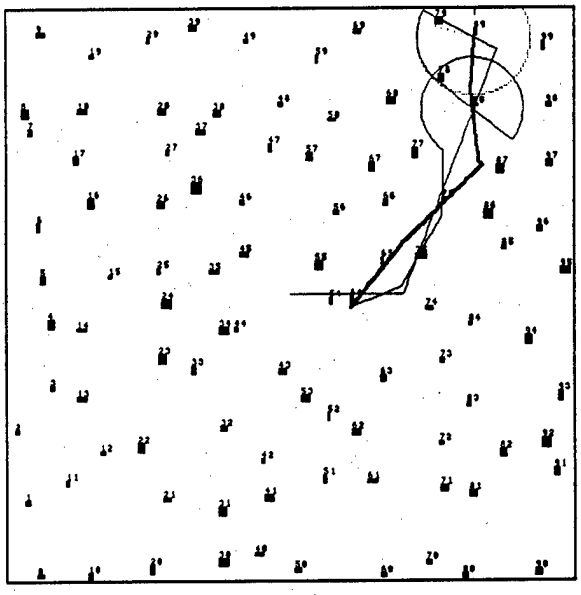

Figure 7.13. Stable Path (thick path) from landmark 64 to 89 after 2nd time.

When the robot traverses back from landmark 64 to 89, it is able to use the viewframes it stored from its previous trip from landmark 89 to 64 to determine a more direct path (Figure 7.13). The robot will determine different paths between the two landmarks, depending upon the direction in which it travels. This is because the robot can not see the same landmarks when traveling in the different directions due to limitations on allowable viewing distance. The further the robot can see, the more direct and similar the paths found under this algorithm become igure 7.14).

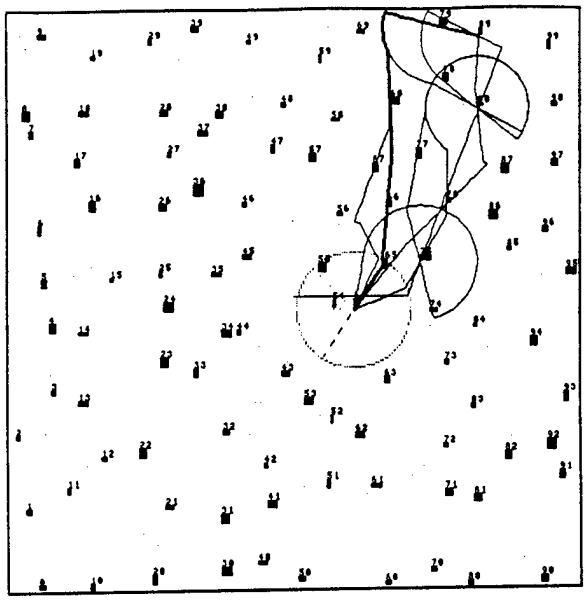

Figure 7.14. Stable Path (thick path) from landmark 89 to 64 after 4th time.

The algorithm has the following steps:

Goal: A landmark with identifier id to go to.

Step 0 If *id* is in current viewframe, go directly to it and the algorithm terminates.

Step 1 Create a *virtual* viewframe V_n containing only the goal *id* with arbitrary orientation. Construct an N by N weight matrix W (N is current total number of viewframes plus one). For each pair of viewframes (V_i, V_j), including the *virtual* viewframe V_n and current starting viewframe V_0, compute $con\ (V_i, V_j)$ by equation (1). If it is greater than a certain threshold (select 0), assign the weight matrix entry $W(i,j)=1$; otherwise $W(i,j)=\infty$. With the weight matrix, one finds a *least* sequence of connected viewframes V_0, V_1, ..., V_n by applying a shortest path algorithm [6]. Alternatively, if the total number of viewframes N is too great, use a breadth-first tree search [6] from V_0 to find adjacent viewframes, such that a viewframe cannot appear twice in a path of the tree.

Step 2 If a sequence of *connected* viewframes are not found, stop. Otherwise the robot performs *viewframe crossing* and *viewframe back-matching* through V_0, V_1, ..., V_n. It walks to the common distinct landmark in both V_0 and V_1.

Step 2.1 If the robot is currently at landmark P of viewframe V_i(i is max), it *viewframe-circles* V_i, i.e. it walks away from P until P is at its visual range-limit, then it circles around P. During the walk, it tests if current viewframe V_c is adjacent to V_m, i.e. $m \in [i+1, n]$ and m is max such that $con(V_c, V_m) > 0$; if m is found, which means a distinct landmark is found, then it changes the direction and walks to the landmark in *both* V_c and V_m.

Step 2.2 Repeat Step 2.1 until the goal is achieved.

7.5 LPB-Based Navigation Without A Compass For Distinct Landmarks

This navigation algorithm assumes distinct landmarks, no compass and use of LPBs. In [21], the robot uses a global map in its spatial memory to indicate each landmark's estimated direction and distance for path planning. Instead of assuming that the robot knows the estimated direction and distance of each landmark in spatial memory, one assumes that the robot only knows the directions of a few selected landmarks called **known landmarks**.

Each pair of the known landmarks forms an **LPB (Landmark-Pair-Boundary) vector** or an **LPB**. LPBs are used to demark visually distinct areas by noting which sides of the LPBs surrounding a region the robot is in. This algorithm uses LPB regions instead of viewframes as the basic descriptions of locations. For an LPB vector $\tilde{l}$ and a location A, we use $\tilde{l}(A)$ to indicate which side of $\tilde{l}$ that A is on. $\tilde{l}(A)$ has 0, 1, 2 values to distinguish different sides. In Figure 7.15, known landmarks K_1, K_2 form LPB $\tilde{l}_{k_1,k_2}$. At A, $\alpha_A > \pi$ (from K_1 to K_2 counterclockwise), $\tilde{l}_{k_1,k_2}(A) = 1$; At B, $\alpha_B < \pi, \tilde{l}_{k_1,k_2}(B) = 0$. At C, it's on the LPB, $\tilde{l}_{k_1,k_2}(C) = 2$. The two landmarks that define an LPB break the LPB into three distinct **LPB segments**.

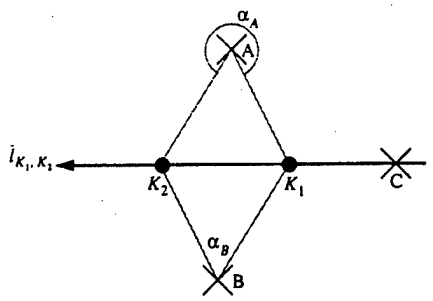

Figure 7.15. LPB (Landmark-Pair-Boundary) Representation.

Suppose there are n known landmarks forming a total of $(N = \binom{n}{2})$ LPB vectors $\tilde{l}_1$, $\tilde{l}_2$, ..., $\tilde{l}_N$. For a set of LPB vectors $\tilde{l}_{k_1}$, $\tilde{l}_{k_2}$, ..., $\tilde{l}_{k_M}$, we define the **LPB projection** for a location A as

$$LPB\_prj(A) = \tilde{l}_{k_1}(A) \bullet \tilde{l}_{k_2}(A) \ldots \tilde{l}_{k_M}(A) \tag{7.2}$$

where ... correspond to string concatenation of the values 0,1, or 2. An **LPB region string** is the LPB projection using the whole set of LPB vectors determined by all the known landmarks. This creates a net of distinct **LPB regions**.

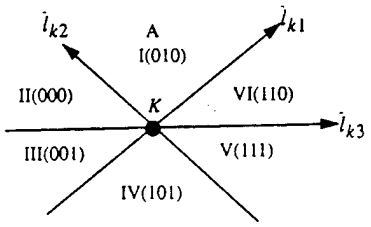

Figure 7.16. LPB Distinct Section Partitions through One Known Landmark.

Path planning involves finding a sequence of LPB segments from the graph created by all the LPB segments formed by known landmarks. The robot walks *along* each LPB and tests both sides of it to see if it is adjacent to the goal region. This requires, at most, $O(n^2)$ LPB vectors to be visited. However, this can be improved. There are a total of $(n-1)$ LPB vectors crossing one known landmark, which will partition the area into, at most, $2(n-1)$ distinct *sections*. Each section is expressed in terms of the LPB projection (onto those LPB vectors) of any location from that section. The basic idea is that the robot goes to the known landmark, walks along parts of two LPB segments, which are borders of the section having the same LPB projection (onto those vectors) as that of goal LPB region. The robot then has at most $O(n)$ LPB vectors to visit. In Figure 7.16, values in parentheses show *distinct LPBs* projections (onto $\tilde{l}_{k1}$, $\tilde{l}_{k2}$, $\tilde{l}_{k3}$) for sections I through VI. To visit region A (section I), the robot only needs to visit parts $K\tilde{l}_{k1}$, $K\tilde{l}_{k2}$ of 2 LPB vectors $\tilde{l}_{k1}$ and $\tilde{l}_{k2}$.

The algorithm has the following steps:

Goal: A given LPB region and corresponding an LPB region string L_g to go to.

Step 0 If any components of L_g is equal to 2 (it is on an LPB vector), the robot first goes to any known landmark on that LPB. It then walks along it in one direction until the goal region is achieved; if so, the algorithm is finished.

Step 1 Initialize segments of each LPB vector as un-visited. Perform **masking** on each known landmark K visited before, i.e., mark the segments of LPB vectors crossing K as visited if they are not borders of the section having the same LPB projection (onto these LPB vectors) as that of L_g. Also initialize the stack SP for the known landmarks as empty.

Step 2 Test the stack SP.

If SP is empty,
  select any known landmark K which is one end
  of an unvisited LPB segment, *push*(K) into SP,
  go to step 2; if K is not found, stop.
Else
  K = *pop*(SP); if K is not one end of any
  non-visited LPB segment, go to Step 2.

Step 3 The robot walks to known landmark K, testing whether the goal region is achieved; if so, the algorithm is finished.

Step 4 If K has not been visited before, the robot performs *masking* on K.

Step 5 If K is one end of non-visited LPB segment S, mark S as visited, $push(K)$ into SP. Else go to Step 2.

Step 6 The robot walks along segment S, testing whether the goal region L_g is found, until one of the following conditions is satisfied:

- If the goal is found, the robot achieves the goal and the algorithm is finished.
- If contradiction to the goal region happens, i.e. , originally the robot is on the same side of one LPB as that of the goal, later different; mark visited for the segment which the robot is heading towards, go to Step 2.
- If the robot arrives at another known landmark K_n, $push(K_n)$ into SP, go to Step 2.

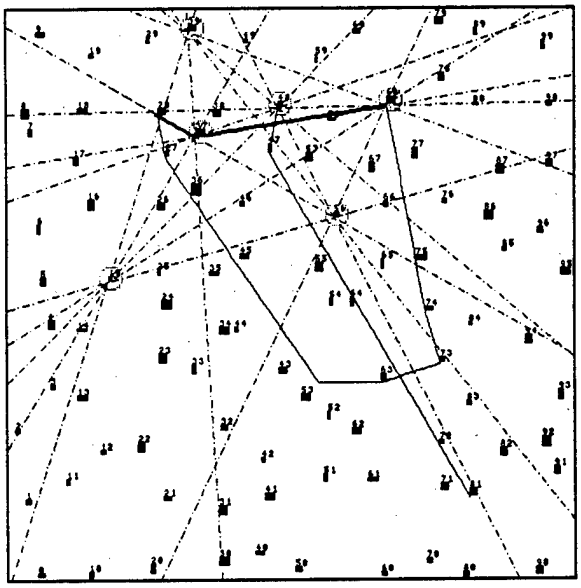

Figure 7.17. Example of Navigation Using LPBs.

Figure 7.17 shows an example of navigation using this algorithm with the viewframes and paths from the previous figures. The known landmarks are circled; the LPB vectors are in a dash-dotted line, and the exploration paths Ψ (solid thin lines) are generated in the same manner as in Figures 7.7, 7.8 and 7.9. The path determined by the robot is shown as a solid thick line from the LPB region near the upper-right corner of the Ψ to the goal region near the upper-left corner of the Ψ. The processing time of this algorithm is $O(n)$, where n is the number of known landmarks. In

addition, experiments have shown the algorithm gracefully degrades as the number of known landmarks is decreased.

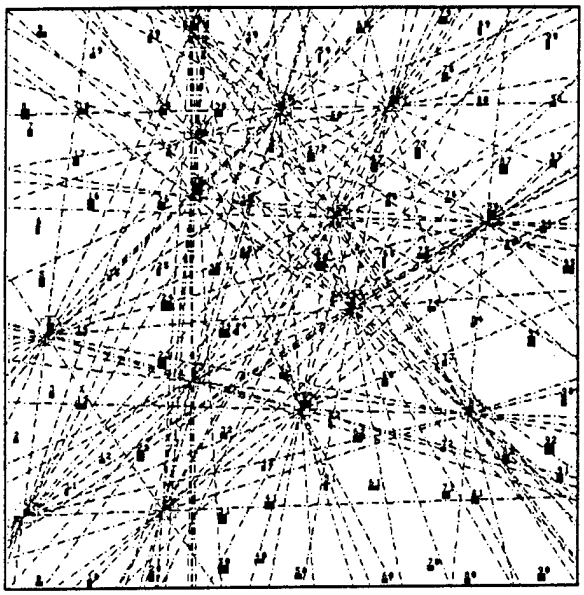

Figure 7.18. LPBs Partitions the Area into Small Regions.

An interesting finding is that if *masking* is applied on all known landmarks as stated in *step 1* of the algorithm, the LPB candidates (i.e. LPB vectors of which LPB masks are not 111) form a **flow** towards the destination region. Figure 7.18 shows LPBs that partition the area into small regions. Figure 7.19 shows the flow towards the goal region near the upper-left corner of the Ψ of Figure 7.17.

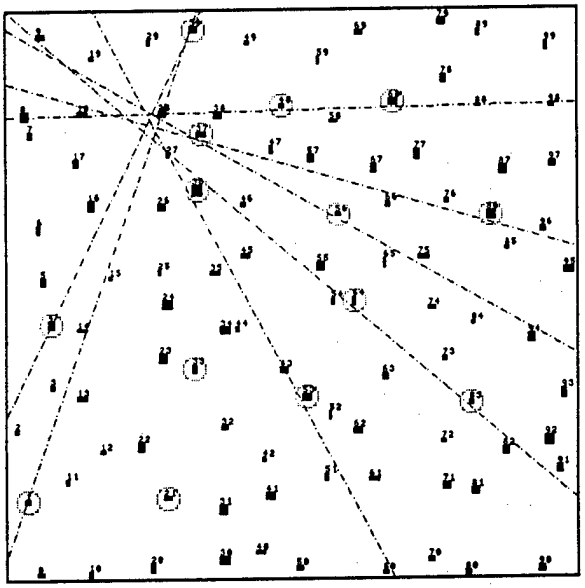

Figure 7.19. LPB Flow Towards the Goal Region Near the Upper-Left Corner of the Ψ of Figure 7.17.

7.6 Navigation Not Using a Compass with a Variable Percentage of Distinct Landmarks

Different alternatives for this case are currently being explored. The characteristic behavior appears similar to navigation using a compass with nondistinct landmarks (hugging to previously explored paths without taking shortcuts), except it is much more sensitive to the allowable viewing distance. One approach for this case is to perform navigation using LPBs defined by landmarks with local-ids. A difficulty is that one or both of the landmarks defining an LPB can disappear as the robot walks away from it. So the LPBs connecting viewframes may not be stable. It may be possible to use a measure of reliability of LPBs between viewframes as a criteria for extracting viewframes.

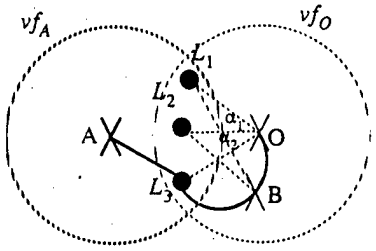

Figure 7.20. Viewframe Back-Matching Without a Compass.

Another approach being investigated when there is a low percentage of distinct landmarks involves modifying *viewframe back-matching* in the algorithm from section 3 to satisfy the constraint of not using a compass. In Figure 7.20, the robot is at A, seeing nondistinct landmarks L_1, L_2, L_3 which are also in vf_O. In order to go near the center O of vf_O to back-match vf_O, the robot first comes to one of L_1, L_2, L_3, say L_3, then it walks along arc L_3BO by maintaining angle $\alpha_2 = \angle L_2OL_3$ walk; and tests if angle $\angle L_1BL_2$ equals α_1, if so, we conclude the robot is close to O. The robot must always see all three landmarks before it comes nearby O (note angles α_1, α_2 are calculated counter-clockwise, so there is only one center). This type of walking is used for *viewframe back-matching without a compass.*

The following algorithm is intended when there is a higher percentage of nondistinct landmarks without a compass. It has the following steps:

Goal: A landmark with local identifier *lcid* (tied to a specific viewframe) to go to.

Step 0 If *lcid* is in current viewframe, go directly to it and the algorithm terminates.

Step 1 Create a *virtual* viewframe V_n containing only the goal *lcid* with arbitrary orientation. Construct an N by N weight matrix W (N is current total number of viewframes plus one). For each pair of viewframes (V_i, V_j), including the *virtual* viewframe V_n and current starting viewframe V_0, if they are at least 3(for $i \neq n$ and $j \neq n$) or 1(for $i = n$ or $j = n$) landmarks with common local ids in both viewframes, one assigns the weight matrix entry $W(i,j) = 1$; otherwise $W(i,j) = \infty$. With the weight matrix, one finds a *least* sequence of connected viewframes V_0, V_1, ..., V_n by applying a shortest path algorithm [6]. Alternatively, if the total number of viewframes N is too great, a

breadth-first tree search [6] from V_0 is used to find adjacent viewframes, such that a viewframe cannot appear twice in a path of the tree.

Step 2 If a sequence of *connected* viewframes are not found, stop. Otherwise the robot performs *viewframe crossing* and *viewframe back-matching* through V_0, V_1, ..., V_n. It walks to the landmark with common local identifier in both V_0 and V_1, where the choice of distinct landmark has priority.

Step 2.1 If the robot is currently at landmark P of viewframe V_i (i is max), it finds two other landmarks with common local identifiers both in current viewframe V_C and vfi, and walks towards the center of vf_i by using *viewframe back-matching without compass*, as explained in Figure 7.20. During the walk, it tests if current viewframe V_C has at least 3 (1 for $m = n$) landmarks with common local identifiers with vf_m, i.e. $m \in [i+1, n]$ and m is max; if m is found, which means three distinct landmarks are found, then it changes the direction to walk to the landmark in *both* V_C and V_m. Otherwise, it performs *back-matching* to V_i; if no ambiguity occurs and the best match is found, the robot updates the local identifiers, i.e. it uses local identifiers in V_i to replace corresponding local identifiers with the same orientations in V_C as well as those local identifiers in V-DB; and then walks to a landmark with common local identifier both in V_i and V_{i+1}.

Step 2.2 Repeat Step 2.1 until the goal is achieved, or failure due to ambiguity.

7.7 Future Work

We have described different range-free qualitative navigation algorithms. The data structures used, especially for the case of nondistinct landmarks, are compatible with the types of features that could be extracted as landmarks with basic image-processing techniques on a robot with a 360 degree field of view. We also have performed experiments to understand path-planning feasibility and efficiency for these algorithms. One measure of path planning efficiency is the ratio of the straight-line distance between two locations compared to the actual distance walked by a robot to go from between the two locations. This measure of efficiency of the compass-based algorithms improves if the number of viewframes, visual range, and distinct landmarks increases. The efficiency of the LPB, non-compass-based algorithm increases as the number of known landmarks increases.

Current work is focusing on navigation using LPBs formed from nondistinct landmarks, viewframe filtering techniques, and different approaches to organizing spatial memory, such as a hierarchical representation of viewframes, along the lines discussed in [12].

Bibliography

[1] Dana H. Ballard and Christopher M. Brown, *Computer Vision.* Prentice Hall, Inc., Englewood Cliffs NJ, 1982.

[2] A. Borning. The programming language aspects of thinglab, a constraint oriented simulation laboratory. *ACM Transactions on Programming Languages and Systems.* 3(4):353ff, 1981.

[3] Terrance E. Boult and Lisa Gottesfeld Brown. Motion segmentation using singular value decomposition. In *Proceedings of the DARPA Image Understanding Workshop,* pages 495-506, January 1992. San Diego CA.

[4] W. Burger and Bir Bhanu. On computing a 'fuzzy' focus of expansion for autonomous navigation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition,* pages 563-568, 1989.

[5] T. Catlin, P. Bush, and N. Yankelovich. Internote: Extending hypermedia framework to support annotative collaboration. In *ACM Hypertext Conference,* 1989.

[6] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* MIT Press, 1990.

[7] R. P. Feynman. *QED: The Strange Theory of Light and Matter.* Princeton University Press, 1987.

[8] C. R. Gallistel. *The Organization of Learning.* The MIT Press, 1990.

[9] J. J. Givson. *The perception of the visual world.* Houghton Mifflin, Boston MA. 1950.

[10] F. G. Halasz, T. P. Moran, and R. H. Trigg. Notecards in a nutshell. In *Proceedings of the ACM CHI + GI Conference on Human Factors in Computing Systems and Graphic Interface,* pages 45-52, April 5-9 1987. Toronto, Canada.

[11] R. Jain. Direct computation of the focus of expansion. *IEEE Transactions on Pattern Analysis and Machine Intelligence,* 5:58-64, 1983.

[12] B. J. Kuipers. Modeling spatial knowledge. *Cognitive Science,* 2:129-153, 1978.

[13] B. J. Kuipers and Y. T. Byun. A qualitative approach to robot exploration and map-learning. In *Proceedings of the Workshop on Spatial Reasoning and Multi-Sensor Fusion,* Los Alton CA, 1987.

[14] Daryl T. Lawton. Constraint-based interence from image motion. In *Proceedings of AAAI-80,* 1980. Stanford CA.

[15] Daryl T. Lawton. Processing translational motion sequences. *Computer Vision, Graphics, and Image Processing,* 22:116-144, 1983.

[16] Daryl T. Lawton and Todd S. Levitt. Knowledge based vision for terrestial robots. In *Proceedings of the DARPA Image Understanding Workshop,* Palo Alto CA, May 1989.

[17] Daryl T. Lawton, Todd S. Levitt, C. C. McConnell, P. Nelson, and J. Clicksman. Terrain models for an autonomous land vehicle. In *IEEE International Conference on Robotics and Automation,* San Francisco CA, April 1986.

[18] D. N. Lee. The optic flow field: The foundation of vision. *Philosophical Transactions of the Royal Society of London, Series B,* 290:169-179, 1980.

[19] W. Leler. *Constraint Programming Languages: Their Specification and Generation.* Addison-Wesley, Reading MA, 1988.

[20] Todd S. Levitt and Daryl T. Lawton. Qualitative navigation for mobile robots. *Journal of Artificial Intelligence,* 1990.

[21] Todd S. Levitt and Daryl T. Lawton. Qualitataive navigation for mobile robots. *Artificial Intelligence,* 44:305-360, 1990.

[22] J. Mundy, P. Vrobel, and R. Joynson. Constraint-based modeling. In *Proceedings of the DARPA Image Understanding Workshop,* May 1989. Stanford CA.

[23] J. K. Ousterhout. An x11 toolkit based on the tcl language. In *Winter USENIX Conference,* 1991.

[24] J. K. Ousterhout. *An Introduction to Tcl and Tk.* Addison-Wesley Publishing, Computer Science Division, Dept. of Elec. Eng. and Computer Sciences at Berkeley, 1993.

[25] C. Ruoff, J. Bowyer, T. Brooks, J. Hanson, K. Holmes, and B. Wilcox. Autonomous ground vehicles: Control system technology development. Technical report, Jet Propulsion Laboratory, Pasadena CA, October 1984.

[26] R. Steeb, S. Cammarata, S. Narain, J. Rothenberg, and W. Giuarla. Cooperative intelligence for remotely piloted vehicle fleet control, analysis, and simulation. Technical report, Rand Corporation, Santa Monica CA, 1986.

[27] Carlo Tomasi and Takeo Kanade. Shape and motion without depth. In *Proceedings of the DARPA Image Understanding Workshop,* pages 258-270, 1990. Pittsburgh PA.

[28] Carlo Tomasi and Takeo Kanade. The factorization method for the recovery of shape and motion from image streams. In *Proceedings of the DARPA Image Understanding Workshop,* pages 459-472, January 1992. San Diego CA.

[29] N. Yankelovich, B. Hann, N. Meyrowitz, and S. Drucker. Intermedia: The concept and construction of a seamless information environment. *IEEE Computer,* 21,1988.